Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jan Makovecki

Objektno-orientirano programiranje s prototipi

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Jurij Mihelič

Ljubljana, 2016

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Objektno-orientirano programiranje z razredi je paradigma, ki že dodobra prežema svet računalništva. Z vzponom programskega jezika javascript pa je doživela razmah tudi manj znana, a podobna paradigma objektnega-orientiranega programiranja s prototipi. V okviru diplomskega dela preglejte področje in ga predstavite. Opišite tudi več programskih jezikov, ki omenjeno paradigmo podpirajo. Primerjajte jezike med seboj glede na osnovne koncepte, ki jih omogoča objektni pristop.

Zahvaljujem se staršem, bratu in prijateljem za moralno podporo ob pisanju diplomske naloge ter mentorju za strokovno pomoč in nasvete, ki so moje delo vodili v pravo smer.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
OOP	object-oriented programming	objektno-orientirano programiranje
API	application programming interface	aplikacijski programski vmesnik
JSON	JavaScript object notation	javascript objektna notacija
XML	extensible markup language	razširljiv označevalni jezik
JIT	just in time	v zadnjem trenutku

Povzetek

Naslov: Objektno-orientirano programiranje s prototipi

Avtor: Jan Makovecki

Diplomska naloga se ukvarja z objektno-orientiranimi programskimi jeziki s prototipi, znanimi tudi kot prototipnimi programskimi jeziki, ter programiranjem v njih. Vsebuje razlago koncepta in komponent klasičnega objektno-orientiranega programiranja z uporabo razredov, predstavitev objektno-orientiranega programiranja s prototipi ter pregled razlik med njima. Drugi del naloge se ukvarja z dejanskimi objektno-orientiranimi programskimi jeziki s prototipi (self, javascript, lua, omega) ter klasičnim, razrednim objektno-orientiranim jezikom za primerjavo (java). Poleg zgodovine ter opisa vsakega od jezikov in njegovih posebnosti naloga vsebuje tudi primere programske kode za vsak jezik. Ti služijo prikazu delovanja posameznega jezika ter primerjavi hitrosti izvajanja različnih pomembnejših operacij v vsakem od jezikov.

Ključne besede: objektno-orientirano, programiranje, objekti, razredi, prototipi, java, self, javascript, lua

Abstract

Title: Object-Oriented Programming with Prototypes

Author: Jan Makovecki

The thesis deals with object-oriented programming languages with prototypes, also known as prototype-based languages, and programming in them. It contains an explanation of the concept and components of classic object-oriented programming with the use of classes, a presentation of object-oriented programming with prototypes and an overview of differences between them. The second part of the thesis deals with actual object-oriented languages with prototypes (such as Self, JavaScript, Lua, Omega) and compares them with a classic, class-based object-oriented language (Java). Alongside the history and the description of each language and its particularities, the thesis also contains examples of code written in each of the languages. Those serve to illustrate the way in which the languages work and compare their performance speed when executing various common tasks.

Keywords: object-oriented, programming, objects, classes, prototypes, java, self, javascript, lua

Poglavje 1

Uvod

Objektno-orientirano programiranje (angl. object-oriented programming) je način programiranja, v katerem so vse komponente programa predstavljene kot objekti, ki hranijo podatke in med seboj komunicirajo ter s tem omogočajo delovanje programov.

Objektno-orientirano programiranje s prototipi (angl. object-oriented programing with prototypes), znano tudi kot na prototipih temelječe programiranje (angl. prototype-based programming), je podvrsta objektno-orientiranega programiranja. Od klasičnega, razrednega objektno-orientiranega pristopa, se razlikuje v tem, da za izdelavo novih objektov ne uporablja razredov, ki vnaprej opišejo sestavo in obnašanje posameznega objekta, pač pa klonira že obstoječe objekte (prototipe) ter jih prilagodi tako, da ustrezajo trenutnim potrebam programa [7, 8].

1.1 Motivacija in cilji

Objektno-orientirano programiranje oziroma objektna-orientiranost na splošno je dandanes pogost izraz, ki ga bolj ali manj upravičeno redno slišimo v opisih raznih programskih jezikov kot pozitivno, sodobno lastnost danega

jezika. Nekateri izmed trenutno najpopularnejših programskih jezikov temeljijo na konceptih objektne-orientiranosti. Programi napisani v njih se uporabljajo praktično na vseh področjih vsakdanjega življenja. Kljub popularnosti takih jezikov pa enotna razlaga koncepta objektne-orientiranosti ne obstaja in mnogim uporabnikom jezikov, ki na njem temeljijo, niti ni jasno, kaj vse naj bi ta izraz sploh zajemal. Razlika med razrednimi in prototipnimi jeziki je še manj poznana in slednje se pogosto umešča kar med prve, kljub temu da so razlike med njimi precejšnje.

Cilj te diplomske naloge je bolje opredeliti in predstaviti koncept objektne-orientiranosti na splošno, nato pa preiti na podrobnejši opis pristopa s prototipi ter opisati razlike med razrednim in prototipnim pristopom tako na praktični kot tudi na idejni ravni. Pri tem se sklicuje na zgodovino nastanka konceptov ter jezikov, njihovo razumevanje ter rabo danes in tudi primere dejanske programske kode. Naloga tako vsebuje opise popularnejših objektno-orientiranih jezikov s prototipi, primere kode zapisane v njih, ter primerjavo med njimi tako z vidika pristopov reševanja različnih problemov kot tudi hitrosti izvajanja podobne kode. Poleg izbranih prototipnih jezikov je v nalogo vključena tudi java, ki služi kot primer popularnega objektno-orientiranega jezika, ki temelji na razredih.

1.2 Pregled vsebine

V Poglavju 2 se posvetimo osnovam objektno-orientiranega programiranja kot pristopa. Opišemo zgodovino nastanka objektno-orientiranega programiranja ter prve jezike, ki so ga uporabljali, nato pa preidemo na nastanek prototipne različice OOP. Dotaknemo se filozofskih razlik med pristopoma ter idej, ki ju ločujejo, nato pa se osredotočimo na splošne koncepte, ki jezik delajo objektno-orientiran, ter jih natančneje definiramo in razložimo.

Poglavje 3 služi pregledu izbranih programskih jezikov, njihovega nastanka in zgodovine, lastnosti, posebnosti ter dela v njih. Vsebuje nekaj primerov kode, da bolje oriše izgled vsakega od jezikov, sicer pa se predvsem osredotoča na posebnosti, zaradi katerih posamezen jezik izstopa.

Poglavje 4 je namenjeno natančnejši primerjavi med jeziki z vidika pristopov ter implementacij različnih konceptov. Mehanizmi v jezikih so za primerjavo izbrani zaradi svoje pomembnosti ali zanimivosti, za vsakega pa je napisan opis delovanja v izbranih jezikih ter primerjava med njimi, vključno z izpostavitvami izjem ter primeri kode za boljši prikaz delovanja. V zadnjem razdelku poglavja preverimo še, kako hitro so posamezni programski jeziki sposobni izvajati nekatere pogostejše operacije. Za vsak test zabeležimo primerjave hitrosti izvajanja ter obremenjenosti procesorja med izvajanjem, v dodatku A pa je za vsakega na voljo tudi programska koda v vseh uporabljenih jezikih.

Poglavje 2

Objektno-orientirano programiranje

2.1 Zgodovina

Prvi, ki je uporabil izraz „objektno-orientirano programiranje“, je bil Alan Kay, ki je leta 1967 ob ukvarjanju s programskim jezikom simula dobil idejo, po kateri je želel izdelati nov sistem programiranja. Ta naj bi temeljil na konceptu objektov, ki jih je v programiranje uvedla simula. Ob izpopolnjevanju ideje ga je vodilo nekaj smernic. O objektih je razmišljal podobno kot o organskih celicah, ki so ločene enote, lahko pa med seboj komunicirajo. Ugotovil je, da bi lahko vsakemu objektu dodelil več algeber, ki bi jih lahko nato povezoval v družine. Želel se je znebiti podatkov kot takih in jih raje predstaviti kot objekte [16].

Vse to je v sedemdesetih letih prejšnjega stoletja vodilo v nastanek bolj dodelanega koncepta objektno-usmerjenega programiranja, z njim pa tudi v nastanek prvega pravega objektno-orientiranega programskega jezika – smalltalka. Celoten začetek zgodovine objektno-orientiranega programiranja je tako tesno povezan z jezikom smalltalk, čeprav je bil glavni namen tega, da bi služil kot jezik za programiranje Dynabooka, prvega resnično uporabnega

osebnega računalnika [22].

Programski jezik simula je obstajal že pred smalltalkom in je pravzaprav uvedel koncepta dedovanja ter objektov, ki sta bila kasneje razpoznana kot glavni lastnosti objektno-orientiranega programiranja, vendar pa takrat sam koncept objektne-orientiranosti še ni bil dodelan. Simulo tako nekateri smatrajo kot bližnjega prednika objektno-orientiranih jezikov [22], medtem ko jo imajo drugi za prvega izmed njih [4].

Po smalltalku se je v sedemdesetih letih objektno-orientirano programiranje razcvetelo in kmalu so začeli nastajati tudi drugi jeziki, ki so temeljili na njegovih konceptih. Danes je objektno-orientirano programiranje eden od najpopularnejših načinov razvoja programske opreme in veliko najpopularnejših jezikov temelji prav na njem.

V začetku devetdesetih let prejšnjega stoletja so se začeli pojavljati programski jeziki, ki so ubrali alternativni pristop k objektno-orientiranemu programiranju in razrede nadomestili s prototipi. Prototipe so si zamislili kot že same po sebi polne objekte (za razliko od razredov, ki so bolj podobni opisom objektov), novi objekti pa v prototipnem pristopu nastanejo s kloniranjem starih. Primeri jezikov, ki uporabljajo prototipe, so predstavljeni v Poglavju 3.

2.2 Filozofski vidik

V primerjavi s klasičnim, razrednim načinom objektno-orientiranega programiranja, je prototipno programiranje konceptualno lažje, kar morda izvira že iz filozofije, na kateri je zasnovano.

Že Platon je obravnaval predstave o objektih drugače, kot primerke teh

objektov v resničnem svetu. Predstave je imel za nekaj več, kot da obstajajo na višji, „bolj resnični“ ravni od naše resničnosti. Razredni objektno-orientirani programski jeziki obravnavajo objekte na podoben način. Razredi so predstave o nekih predmetih sveta, splošne definicije sestave in obnašanja posameznega tipa objektov. Objekti so primerki teh predstav v resničnosti, konkretni predmeti, ki vsebujejo dejanske vrednosti in rešujejo dejanske naloge.

Filozofija prototipnega objektno-orientiranega programiranja se ne zanaša toliko na klasifikacijo objektov in iskanje neke, ene prave predstave o njih, ampak bolj stremi k temu, da bi objekte predstavila kot čim bolj oprijemljive in intuitivne. Pogost argument v prid temu pristopu je, da se ljudje lažje učimo iz primerov kot iz abstraktnih idej. Nov objekt tako ne nastane iz razreda, neke višje ideje o njem, ampak iz drugega, že obstoječega objekta kot nova različica narejena po primeru, ki jo nato lahko prilagodimo našim potrebam. Tak način dela je bolj intuitiven in izgleda, da je bližje človeškemu načinu razmišljanja [23].

2.3 Pregled konceptov

Vsaka komponenta v objektno-orientiranem jeziku je objekt, ki v kombinaciji z drugimi objekti tvori program. Na zunaj so si vsi precej podobni – imajo ime, s katerim jih identificiramo, sposobnost komunikacije z ostalimi objekti ter v sebi neka stanja in zmožnosti obdelave podatkov, ki so skriti pred zunanjim svetom. Objekt ima lahko tudi druge lastnosti in funkcije, a te zunanjemu opazovalcu običajno niso vidne, in samo od objekta je odvisno, če mu jih bo pokazal.

Dandanes obstaja veliko programskih jezikov, ki so bolj ali manj objektno-orientirani oziroma podpirajo objektno-orientiran način programiranja. Im-

plementacije objektov se tako od jezika do jezika lahko precej razlikujejo. Ne obstaja nek splošen sporazum o konceptih, ki naj bi se jih objektno-orientiran jezik držal. Kljub temu lahko najdemo in naštejemo peščico pomembnejših, s katerimi se večina strinja [2].

2.3.1 Dedovanje

Prvi med njimi je koncept dedovanja (angl. inheritance), ki sega v preteklost še pred čas objektno-orientiranih programskih jezikov, saj je bil predstavljen že v jeziku simula leta 1967 [2]. Simula je najbližji predhodnik jezika smalltalk [22], v katerem je bil kasneje koncept dedovanja še dodelan ter privzet kot eden od osnovnih konceptov objektno-orientiranega programiranja. Adam Kay kot začetnik objektno-orientiranega programiranja dedovanja sicer ni smatral za nujnega [16], vseeno pa je ta koncept postal splošno sprejet ter celo citiran kot edina posebnost, s katero je objektno-orientirano programiranje sploh prišlo na dan [2].

Definicije dedovanja v sklopu objektno-orientiranega programiranja se sicer razlikujejo, vendar lahko iz večine razberemo, da gre za *mehanizem, ki omogoča, da so podatki in obnašanje enega objekta vključeni v oziroma uporabljeni kot osnova za drug objekt* [2].

Več o dedovanju in načinih njegove implementacije je zapisano v primerjavi jezikov v Poglavju 4.2.

2.3.2 Objekt

Objekt (angl. object), prvič predstavljen v programskem jeziku simula, lahko na splošno definiramo kot *individualen, nedoločljiv predmet, lahko je resničen ali abstrakten, ki vsebuje podatke o sebi ter opise svojih manipulacij s podatki*

[2]. Najpogosteje je predstavljen kot primerek (angl. instance) nekega razreda, lahko pa tudi kot kopija prototipa ali na kakšen tretji način.

2.3.3 Razred

Objektno-orientirano programiranje z razredi (angl. class) je le eno od vrst objektno-usmerjenega programiranja, vendar pa je izmed njih prvo ter daleč najpopularnejše, tako da se razred pogosto smatra kot osnovni koncept slednjega. Tako kot objekt izvira iz programskega jezika simula, definiramo pa ga lahko kot *opis organizacije in dejanj, ki si ga deli eden ali več podobnih objektov* [2]. Razred lahko v praksi vidimo kot nekakšen načrt za objekt – vanj se zapiše definicija objekta, po kateri se objekt nato zgradi (instancira) ter ponavadi napolni s podatki. Tako dobljen posamezen objekt se lahko nato uporabi v programu.

2.3.4 Enkapsulacija

Za koncept enkapsulacije (angl. encapsulation) je težko najti enotno definicijo, saj so mnenja o njej precej različna. To je delno tudi razlog, da ni točno določeno, kdaj in kje se je koncept prvič pojavil. Vseeno je enkapsulacija pomemben del objektno-orientiranega programiranja, kot njeno približno definicijo pa bi lahko razumeli *tehniko za načrtovanje objektov, ki omejuje dostop do podatkov objekta ter njegovo obnašanje tako, da določi omejeno množico sporočil, ki jih objekt te vrste lahko sprejme* [2].

2.3.5 Metoda

Metode (angl. method) so eden najpomembnejših gradnikov objektno-orientiranih programskih jezikov, saj opravljajo tako manipulacijo s podatki kot komunikacijo med objekti. Formalnejšo definicijo bi lahko zapisali kot *način*

za dostopanje do informacij nekega objekta, njihovo nastavljanje ali pa manipuliranje z njimi [2]. Metode so se najprej pojavile v programskem jeziku smalltalk kot procedure, ki so nerazdružljivo vezane v objekte.

2.3.6 Podajanje sporočil

Ker so objekti ločene enote, je za delovanje programa nadvse pomembno, da obstaja način, na katerega lahko med seboj komunicirajo. Objektno-orientirani programski jeziki morajo tako vsebovati način za podajanje sporočil (angl. message passing), definiran kot *proces, preko katerega objekt pošlje podatke drugemu objektu, ali pa drugi objekt prosi, naj izvede neko metodo* [2]. To je ponavadi realizirano preko klicev metod danega oziroma drugih objektov.

2.3.7 Polimorfizem

Polimorfizem (angl. polymorphism) bi lahko definirali kot *zmožnost različnih razredov, da odgovorijo na isto sporočilo, pri čemer vsak implementira potrebno metodo sebi primerno* [2]. Tudi tu je definicija splošnejša, saj gre za pojem, ki si ga različni programski jeziki razlagajo precej različno.

2.3.8 Abstrakcija

Abstrakcija (angl. abstraction) je v glavnem razumljena kot *poenostavljanje kompleksnih podrobnosti podatkov ter njihova predstavitev z lažjim modelom, ki omogoča večjo preglednost in lažje razumevanje* [2]. Nekateri jo opisujejo tudi kot odstranjevanje razlik med objekti, da lažje vidimo, kaj jim je skupno. V obeh primerih gre za postopek poenostavljanja za lažje opažanje tega, kar je pomembno, mimo ne tako važnih podrobnosti.

2.3.9 Razvrstitev

Sedaj, ko imamo identificirane in naštete osnovne koncepte objektno-orientiranega programiranja, lahko poskusimo ugotoviti povezave med njimi ter jih umestiti v neko celovito razvrstitev (angl. taxonomy), s katero lahko predstavimo objektno-orientirano programiranje. Morda je najbolj smiselno, da naštete koncepte razdelimo med dva konstrukta: strukturo (angl. structure) in obnašanje (angl. behavior) [2]:

- *Struktura* zajema koncepte abstrakcije, dedovanja, enkapsulacije, objektov ter razredov. Osredotoča se predvsem na glavne gradnike objektno-orientiranega programiranja, njihovo zgradbo ter odnose med njimi.

- *Obnašanje* zajema metode, podajanje sporočil ter polimorfizem. Osredotoča se predvsem na procese ter komunikacijo v sistemu.

2.3.10 Ostalo

Poleg naštetih izrazov ter opisanih konceptov sodijo na področje objektno-orientiranega programiranja še mnogi drugi. Izmed izpuščenih so gotovo najvidnejši člani (angl. attributes) posameznih objektov, ki so ohlapno rečeno podatki različnih oblik, hranjeni znotraj objektov in tako nepogrešljiv del njihove strukture. Za razredno objektno-orientirano programiranje je pomembno tudi instanciranje, ki je kreacija primerkov objektov iz njihovih razredov. Skrivanje informacij je koncept, ki bi ga lahko delno uvrstili pod enkapsulacijo, čeprav je pojem sam precej širši. Kot rečeno, je konceptov še več, a težko je reči, kateri izmed nenaštetih še spadajo med glavne, kateri pa ne.

Poglavje 3

Izbrani programski jeziki

To poglavje je namenjeno pregledu ter predstavitvi izstopajočih objektno-orientiranih jezikov s prototipi ter jave kot primera klasičnega objektno-orientiranega jezika. Opisi zajemajo kratko zgodovino jezikov ter okoliščine njihovega nastanka, njihov osnovni namen, način delovanja, sestavo in delovanje njihovih objektov, pregled dedovanja ter možne dodatne posebnosti, ki jih je pri posameznem jeziku vredno omeniti. Poleg tega je vsakemu od (dandanes še relevantnih) jezikov dodan še primer preprostega programa, napisanega v njem, za predstavo o izgledu njegove programske kode.

3.1 Java

Program 3.1: FizzBuzz v javi

```
public class FizzBuzz {
    public static void main(String args[]) {
        for (int i = 1; i <= 100; i++) {
            if (i % 3 == 0) System.out.print("Fizz");
            if (i % 5 == 0) System.out.print("Buzz");
            if (i % 3 != 0 && i % 5 != 0) {
                System.out.print(i);
            }
            System.out.println();
        }
    }
}
```

V zgodnjih devetdesetih letih so pri podjetju Sun Microsystems razvili programski jezik java, ki so ga zasnovali kot preprostejšo verzijo jezika C++ [12]. Java se je razširila kot jezik za snovanje spletnih aplikacij, danes pa je eden najpopularnejših jezikov na svetu uporabljan na mnogih področjih, vse od mobilnih sistemov, do spletnih strežnikov, iger itd.

Java je edini jezik, vključen v to diplomsko nalogo, ki temelji na klasičnem konceptu objektno-orientiranega programiranja z razredi in ne uporablja prototipiranja. Njegova naloga v tem delu je, da služi kot primer splošnega, sodobnega objektno-orientiranega jezika z razredi, tako za primerjavo pristopov kot tudi hitrosti delovanja. Za to nalogo je bila izbrana zaradi svoje popularnosti, dodelanosti ter dobre definiranosti jezika, kot tudi hitrega delovanja, zahvaljujoč njenemu JIT prevajalniku (angl. just-in-time compiler, prevajalnik v zadnjem trenutku). Ta skrbi za to, da so v trenutku pred zagonom programa računsko zahtevne oziroma pogosto klicane metode prevedene iz vmesne naravnost v strojno kodo in se tako lahko izvajajo dosti hitreje.

V programu 3.1 lahko vidimo preprost primer kode, spisane v javi. Programček je FizzBuzz, preprosta uganka, s katero ponekod testirajo zelo osnovno razumevanje programerskih pristopov. Naloga programa je, da izpiše števila od 1 do 100, pri čemer nadomesti števila, deljiva s 3, s „Fizz“, števila, deljiva s 5, z „Buzz“, števila, deljiva z obema, pa s „FizzBuzz“. Primer je bil za nalogo izbran, ker na kratko prikaže sintakso zank, vejitev, računskih operacij, primerjav vrednosti ter izpisa besedila v posameznem jeziku. V tem primeru v javi vidimo na začetku še deklaracijo javnega razreda `FizzBuzz` (več o zasebnosti ter nadzoru dostopa v Poglavju 4.4) ter glavne metode `main`, s katero se začne izvajanje javanskih programov.

Ker je java klasični objektno-orientiran jezik z razredi, so objekti, ki jih v njej uporabljamo, primerki razredov. Objetki vsebujejo metode ter statično tipizirane spremenljivke, njihove sestave pa po kreaciji ni več mogoče spreminjati. Dedovanje v javi je implementirano s klasičnim nasledstvom (angl. inheritance) in prav tako uporablja razrede. Če želimo ustvariti razred, ki podeduje lastnosti nekega drugega razreda, ga iz tega drugega izpeljemo in dobimo podrazred (angl. subclass). Razred, iz katerega smo izpeljevali, se sedaj imenuje nadrazred (angl. superclass). V javi ima vsak razred samo en, točno določen nadrazred. Če mu tega ne določimo sami, se privzeto nastavi na osnovni razred `Object`. Podrazrede se v javi ustvari z besedo `extends` pri deklariranju novega razreda (recimo `public class Podrazred extends Nadrazred`). Podrazred od nadrazreda podeduje vse metode in spremenljivke razen zasebnih (več o teh v poglavju 4.4) [18].

3.2 Self

Program 3.2: FizzBuzz v selfu

```
_AddSlots: (| x <- 0 |).
[x < 100] whileTrue: [
    x: x + 1.
    ((x % 3) = 0) ifTrue: ['Fizz' print].
    ((x % 5) = 0) ifTrue: ['Buzz' print].
    (((x % 3) != 0) && ((x % 5) != 0)) ifTrue: [x print].
    '\n' print.
]
```

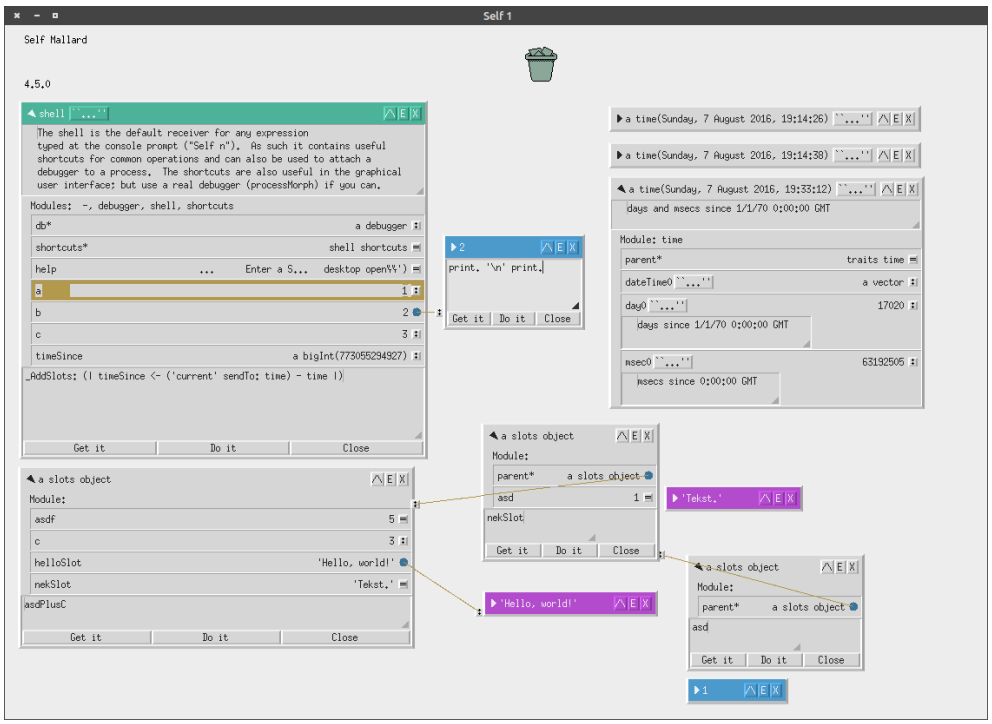
Programski jezik self sta leta 1986 načrtovala David Ungar in Randall B. Smith pri laboratorijih Xerox PARC – na istem mestu, kjer je bil razvit smalltalk. Delo na njem se je nadaljevalo na stanfordski univerzi, kasneje pa se je projekt preselil v Sun Microsystems laboratorije, kjer je delo na jeziku teklo do leta 1995 [1]. Za tem se je jezik razvijal kot odprtokoden projekt in kot tak obstaja še danes.

Self je bil že v osnovi zastavljen bolj eksperimentalno – z njegovim razvojem so ugotavljali, kaj vse je mogoče z objektno-orientiranim načrtovanjem in prototipi narediti in doseči. Doživel je veliko različnih implementacij ter tekel na raznih platformah (od Open Solaris do Android x86), vendar je šele z verzijo 4.0 in novim grafičnim okoljem postal primernejši za „resno programiranje“. Kljub temu dandanes ne uživa podobne priljubljenosti in razširjenosti kot javascript ali lua, ostaja bolj akademsko usmerjen.

Self se od večine dandanes popularnih programskih jezikov razlikuje že v načinu dela. Jezik sestoji iz navideznega stroja, na katerem teče, ter grafičnega uporabniškega vmesnika, ki ga predstavi uporabniku. Selfovo grafično delovno okolje se imenuje „namizje“ in njegov osnovni namen je uporabniku objekte predstaviti karseda neposredno in oprijemljivo. Namizje

je v celoti napisano v selfu, uporabnik lahko tako njegove komponente in njihov izgled med delom prilagaja in spreminja po želji ter tako upravlja z objekti in njihovimi lastnostmi.

Namizje sestavljajo ozadje ter objekti na njem, ki so podobni oknom, kot jih uporabljajo današnji operacijski sistemi. Okna so sestavljena iz več razdelkov, med katerimi lahko najdemo tekstovna polja, ukazne lupine, prikazovalnike slik ter celo preprost spletni brskalnik. Tudi kodo za FizzBuzz, napisano zgoraj (program 3.1), je za izvajanje potrebno vnesti v ukazno lupino znotraj self namizja. Koda bo v objekt, iz katerega jo poženemo, dodala dodatno režo `x`, potrebno za izvajanje zanke. Izpis programa se bo zgodil v izhod navideznega stroja, torej izven self namizja, v ukazni vrstici, iz katere smo namizje zagnali.



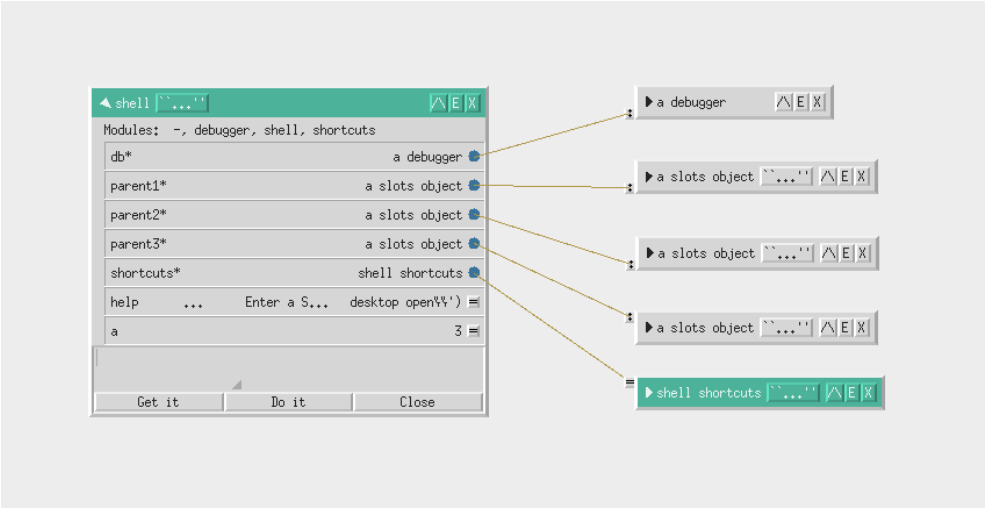
Slika 3.1: Namizje self

Na Sliki 3.1 je prikazano self namizje z več okni. Levo zgoraj vidimo ukazno lupino (angl. shell), ki se v namizju tudi odpre kot prvi objekt. Deluje kot vsi ostali objekti in mogoče ji je dodati reže – na sliki ima dodane reže `a`, `b` in `c` ter režo timeSince, ki smo jo dodali s kodo, ki je še vidna v ukazni lupini. Zapisana koda doda režo `timeSince` objektu, v katerem je pognana, v režo pa shrani razliko milisekund med časom klica ter osmim februarjem 1992. Reža `b` je prikazana kot lasten objekt, povezan na svojo režo v lupini, vanj pa je vpisana koda, ki izpiše ime objekta ter novo vrstico. Desno zgoraj vidimo več časovnih objektov, levo spodaj pa se nahaja navaden objekt z režami (angl. a slots object) z več komponentami. Iz njega je izpeljan še en objekt z režami, ki preko polja `parent*` kaže na prvega in ga s tem določa za svojega starša. Tudi iz drugega objekta je izpeljan objekt, ki podobno kaže nanj kot na svojega starša. Poleg drugega in tretjega objekta lebdita še vsebini rež `nekSlot` ter `asd` prvega objekta, ki sta bili poklicani iz objektov, poleg katerih se nahajata, kot prikaz dostopa do podedovanih komponent.

Tekstovna polja ponavadi prikazujejo vsebino rež (angl. slots), ki so dejanske sestavine objektov in jih lahko na kratko opišemo kot pare imen in vrednosti. Vsebina rež so lahko reference na druge objekte, lahko pa tudi enostavnejše komponente – denimo števila, besedilo ali pa programska koda. Čeprav se reže načeloma ločijo na podatkovne in izvedljive (ter te še naprej na navadne ter bločne metode), pri dostopu do objekta ni pomembno, kaj je v reži, na katero se sklicujemo. Ko objekt prejme sporočilo z imenom reže, izvede kodo v njej in vrne rezultat ali pa preprosto vrne njeno vsebino.

Self se za delovanje programov močno drži načela pošiljanja sporočil med objekti. Zanimiv primer tega je, da nima posebne operacije za prirejanje vrednosti – vrednosti rež se spreminja s pošiljanjem sporočil. Reže, katerih vsebino je mogoče spreminjati, imajo tako sestrske reže, ki prirejajo vrednost prvim. Ko se v programu zgodi prirejanje vrednosti, denimo reži „x“, se vrednost kot sporočilo pošlje sestrski reži, „x:“, ki nato to vrednost vpiše v režo

x [1].



Slika 3.2: Dedovanje v programskem jeziku self

Self podpira večkratno dinamično dedovanje preko starševskih rež (označenih s simbolom * za imenom) ter delegacije [1]. Vsakemu objektu je lahko dodeljenih ena ali več starševskih rež, v katere se zapiše kazalce na objekte, ki služijo kot starši trenutnega. Ko se v kontekstu objekta pojavi zahteva po reži, ki je objekt ne pozna, to zahtevo delegira naprej svojim starševskim objektom. To je tudi edini način, na katerega so poleg pošiljanja sporočil objekti v selfu povezani – kopija objekta je namreč neodvisna od njegovega originala (prototipa) in kakršnekoli spremembe na njej na slednjega nimajo vpliva. Na sliki 3.2 lahko vidimo, kako izgleda dedovanje v programskem jeziku self. Prikazan je objekt `shell`, ki vsebuje reže `db*`, `parent1*`, `parent2*`, `parent3*` in `shortcuts*`. V vseh teh režah so zapisane povezave na starše objekta `shell`, kar nam povedo zvezdice (*) na koncih njihovih imen. Konce povezav, ki se dotikajo staršev, lahko kadarkoli z miško potegnemo na kak drug objekt in s tem zamenjamo objekte, od katerih `shell` deduje.

3.3 Javascript

Program 3.3: FizzBuzz v javascriptu

```
for (var i = 1; i <= 100; i++) {
    var out = "";
    if (i % 3 == 0) out = out + "Fizz";
    if (i % 5 == 0) out = out + "Buzz";
    if (i % 3 != 0 && i % 5 != 0) out = i;
    console.log(out);
}
```

Leta 1995 je Brendan Eich v desetih dneh spisal prototipni programski jezik, tedaj imenovan Mocca [6]. Jezik je bil kmalu (večinoma zaradi tedanje popularnosti jave) preimenovan v javascript, ime, pod katerim ga poznamo še danes. Na njem temelji specifikacija skriptnega jezika ecmascript, katere primarni namen je spletno programiranje na strani odjemalca, javascript pa se je skozi več preobratov uspel ohraniti kot najpopularnejša implementacija ecmascripta. Čeprav je bil na začetku javascript mišljen kot skriptni jezik, je od tedaj že zdavnaj prerasel v splošno namenski jezik [10], ki dandanes ni le standard za spletno programiranje na strani odjemalca, pač pa je široko uporabljan tudi na drugih področjih in slovi kot najbolj razširjen objektno-orientiran programski jezik s prototipi na svetu.

Danes je implementacij javascripta več, saj brskalniki pogosto razvijajo lastne in strogo gledano je pravi javascript le verzija, ki jo uporablja Mozilla v brskalniku FireFox [10]. V praksi se tudi ostale implementacije kliče kar javascript. Čeprav obstajajo med njimi razlike, tako v delovanju kot v hitrosti izvajanja, se vse bolj ali manj držijo ecmascript standardov in ista koda deluje v različnih brskalnikih enako.

Javascript se je ob nastanku zgledoval po več jezikih. Od selfa je ja-

vascript prevzel prototipni pristop k programiranju [10]. Od jave je poleg imena delno prevzel tudi programsko sintakso – podobna sta si v obliki deklaracij funkcij, zank in vejitev, uporabi zavitih oklepajev za označevanje blokov kode znotraj omenjenih konstruktov, postavljanju podpičij na konce vrstic (čeprav to v javascriptu ni vedno nujno, pogosto zna zaključene vrstice razpoznati tudi sam) itd. Podobnost javi lahko v praksi vidimo v FizzBuzz programu 3.3, ki je sintaktično precej podoben javinemu (3.1). Razlika med primeroma je v tem, da v javascriptu ni potrebno deklarirati razreda (teh tako ne uporablja) ter glavne metode – program se preprosto začne izvajati na začetku. Poleg tega ukaz `console.log()` vedno izpisuje v novo vrstico, zato je potrebno Fizz in Buzz shranjevati v spremenljivko, da se lahko pri številih, deljivih s 15, najprej združita in ju lahko nato izpišemo naenkrat.

Kot večina drugih prototipnih jezikov je javascript dinamično tipiziran. Spremenljivke v njem so netipizirane – tudi če spremenljivki na začetku dodelimo vrednost enega tipa, ji lahko kasneje brez problema dodelimo vrednost drugega tipa. Če v javascriptu izvedemo operacijo, denimo seštevanje, med različnima tipoma spremenljivk (ali konstant), bo poskusil pretvoriti eno v drug tip ter operacijo vseeno izvesti. V primeru seštevanja števila in besedila bo število pretvorjeno v besedilo in priključeno k preostalemu besedilu. Tudi če uporabimo operator, ki ne podpira določenega tipa podatkov, se bo zgodila pretvorba. Množenje dveh števil, zapisanih kot besedilo (recimo znotraj dvojnih narekovajev), pretvori obe v dejanski števili in ju nato zmnoži v novo dejansko število [10]. Vsak tip je mogoče pretvoriti tudi v logično vrednost, pri čemer je resnično vse, razen izrecne besede neresnično (angl. false), nedefinirane spremenljivke, vrednosti null, praznega niza, pozitivne ter negativne ničle ter vrednosti `NaN`, ki predstavlja neveljavno število.

Objekti v jeziku javascript so sestavki iz več različnih neurejenih vrednosti, do katerih lahko dostopamo preko tekstovnih ključev. Ustvariti jih je mogoče na tri načine: lahko uporabimo dobesedni opis objekta (angl. object

literal), kjer naštejemo vse pare ključev ter vrednosti in jih ločimo z vejicami, lahko jih ustvarimo z besedico `new`, nov, ki ji sledi konstruktorska funkcija z opisom komponent objekta, lahko pa uporabimo tudi funkcijo za ustvarjanje objektov `Object.create`. Po stvarjenju se z objekti upravlja preko referenc na njih, njihova dinamična narava pa omogoča, da se jim lahko lastnosti (angl. properties) oziroma komponente skoraj kadarkoli dodaja ali odvzema. Tudi posamezni komponenti lahko določimo več lastnosti:

ali se sme vanjo pisati (angl. writable), torej če je njeno vrednost mogoče nastaviti,

ali se jo da oštevilčiti (angl. enumerable), kar pove, da se njeno ime vrne v for/in zanki, ki v splošnem vrača komponente neke strukture in

ali se jo da nastavljati (angl. configurable), torej če je mogoče njene lastnosti spreminjati ali pa jo kar izbrisati [10].

Javascript objekte lahko serializiramo in zapišemo v javascript objektni notaciji, bolj znani pod kratico JSON (angl. JavaScript Object Notation). Ker je ta format zapisa dovolj pregleden in bolj kompakten kot XML, saj ne uporablja začetnih in končnih značk, se je njegova uporaba razširila tudi izven jezika javascript. Postal je splošno namenska notacija za shranjevanje in izmenjavo podatkov [15]. Primer JSON zapisa preprostega objekta lahko vidimo v programu 3.4.

Program 3.4: JSON zapis objekta

```
{   "kategorija" : "pribor",
    "prevzem" : "osebno",
    "blago" : [
        {"predmet" : "zlica",   "kosov" : 30 },
        {"predmet" : "vilice",  "kosov" : 30 },
        {"predmet" : "noz",     "kosov" : 25 } ]
}
```

Vsak objekt ima lahko prototip, čigar vrednosti so mu na voljo, privzeto pa je prototip vsakega novega objekta objekt `Object`. Če objekt ustvarimo z uporabo funkcije `Object.create()`, lahko kot prvi argument podamo nek drug objekt, ki bo služil kot prototip novemu. Podobno lahko prototip definiramo v konstruktorski funkciji tako, da ga priredimo vrednosti `[objekt].prototype`. Če na tem mestu napišemo null, bo nov objekt ustvarjen ex-nihilo, torej iz ničesar, brez prototipa in kakršnihkoli lastnosti, kar je v javascriptu sicer redko uporabljano, a mogoče [10].

Dedovanje v jeziku javascript je implementirano preko delegacije. Če želimo dostopati do vrednosti nekega ključa, ki ga v danem objektu ni, se zahteva po njem delegira naprej prototipu tega objekta. Če ključa tudi prototip nima, se zahtevo posreduje njegovemu prototipu in tako naprej do null. Če želimo prirediti objektu neko vrednost s ključem, ki v njem še ne obstaja, se ta ustvari na novo, na prototipe pa to nima nobenega vpliva. Edina izjema pri tem je, da ni mogoče ustvariti vrednosti s ključem, ki v verigi prototipov že obstaja in je bil definiran z izključno bralnim dostopom [10]. Ker ima vsak objekt lahko le en prototip, javascript ne podpira večkratnega dedovanja. Če želimo v objekt vključiti funkcije iz več objektov, lahko uporabimo mešane objekte (angl. mixins), ki v enem objektu združijo komponente iz več drugih. Za to lahko uporabimo na primer metodo `Object.assign()`, ki v dani objekt kopira komponente drugih objektov v bolj konkatenacijskem pristopu [3]. Tako dobimo en objekt, ki je kompozicija željenih komponent in ga lahko uporabimo kot prototip za naše objekte, ki bodo na ta način podedovali vse potrebne komponente.

3.4 Lua

Program 3.5: FizzBuzz v lui

```
for i=1,100 do
    if i % 3 == 0 then io.write "Fizz" end
    if i % 5 == 0 then io.write "Buzz" end
    if i % 3 ~= 0 and i % 5 ~= 0 then io.write(i) end
    print ""
end
```

Lua je skriptni, dinamično tipiziran jezik. Njen razvoj se je začel leta 1993 na Pontifiški katoliški univerzi v Riu de Janeiru. Od tedaj je dosegla svetovno razpoznavnost in široko uporabo na različnih področjih – vse od spleta do razvoja iger in vgrajenih sistemov.

Za razliko od ostalih dandanes zelo popularnih jezikov je lua edini, ki izvira iz države v razvoju, Brazilije, ki se je v začetku devetdesetih let trudila biti čimbolj samostojna. Ker na področju računalništva niso želeli biti odvisni od drugih držav, so veljale omejitve trgovanja s strojno in programsko opremo – podjetja so morala dokazati, da v Braziliji ne morejo dobiti potrebnih orodij za svoje delovanje, da jim jih je bilo dovoljeno kupiti drugod. Tako je bilo veliko programske opreme za tamkajšnjo uporabo razvite v Braziliji in tudi lua je bila produkt tega okolja – njeni snovalci so potrebovali skriptni jezik in ker tedaj niso našli primerne alternative, so se odločili razviti lastnega [14].

Lua je jezik, ki je bil že od začetka načrtovan tako, da bi omogočal čim večjo razširljivost ter možnost integracije z drugimi programskimi jeziki. V osnovi je lua napisana v C-ju in za svoje delovanje uporablja veliko C-jevskih knjižnic, ponuja pa tudi C API, ki dodajanje teh še olajša. Njene C-jevske korenine dajejo lui veliko prenosljivost – deluje na praktično vseh platformah,

za katere obstaja C-prevajalnik. Poleg C-ja sodeluje lepo tudi z drugimi jeziki, kot so fortran, java, smalltalk in ada, kar je ena njenih večji prednosti [13].

Lua je zelo majhen in preprost jezik, njena celotna programska koda ne presega velikosti diskete. Ker temelji na majhnem številu konceptov in namerno uporablja preprosto sintakso s precej angleškimi besedami (kot vidimo v programu 3.5), se je tudi ni posebej težko naučiti, poleg tega pa omogoča programiranje v več paradigmah in se tako lahko dobro prilagodi dani nalogi. Poleg prototipnega programiranja, na katerega se bomo osredotočali mi, podpira med drugim tudi objektno-orientirano programiranje z razredi ter funkcijsko programiranje.

Objekti v lui so predstavljeni kot asociativne tabele. Tabele so pravzaprav edina podatkovna struktura, ki jo lua pozna in uporablja [13]. Zaradi luinega dinamičnega tipiziranja lahko v tabelah brez težav sobivajo vrednosti različnih podatkovnih tipov, kot recimo števila, besedilo in reference na objekte. Podatki v tabelah so predstavljeni kot pari vrednosti in ključev, preko katerih se do vrednosti dostopa, pri čemer so lahko celo ključi iste tabele različnih tipov. Za „navadno“ tabelo, indeksirano s števili, lua tako uporabi številke kot ključe, vseeno pa je dodati polje komentar, ki razloži vsebino tabele, v lui povsem legalna operacija. Pri splošnih objektih so ključi imena komponent, torej besedilo. Celo knjižnice, ki jih lua uporablja, so v osnovi samo tabele in jih lahko kot take tudi uporabljamo – na primer, če želimo izvedeti imena funkcij, ki jih vsebuje dana knjižnica. Ker vemo, da je knjižnica tabela, lahko nad njo poženemo operacijo za izpis ključev tabele. Rezultat, ki nam ga vrne lua, so imena vseh komponent izbrane knjižnice.

Ta zadnji primer je mogoč, ker so v lui tudi funkcije tretirane kot navadne spremenljivke. Mogoče jih je podajati kot argumente, vračati iz drugih funkcij ter jim poljubno spreminjati vrednost. To omogoča več zanimivih operacij,

saj tudi osnovne funkcije (kot na primer `print`, funkcija za izpis besedila) niso nobena posebnost – mogoče je na novo definirati njihovo obnašanje ali pa celo nastaviti njihovo vrednost na `nil`, prazno, če jih želimo popolnoma onemogočiti.

Dedovanje se v lui izvaja preko posebnega polja v tabelah, imenovanega `__index`. Glede na to, kaj v polje `__index` vstavimo, lahko lua podpira tako enostavno, enojno dedovanje kot tudi večkratno dedovanje, z več različnimi predniki oziroma prototipi. Enojno dedovanje je preprosto – v polje `__index` vstavimo referenco na objekt, ki naj deluje kot starš trenutnega objekta. Če želimo nato klicati polje, ki v danem objektu ne obstaja, lua prebere referenco na starševski objekt iz polja `__index` ter nadaljuje iskanje v njem. Če tudi v staršu iskanega polja ne najde, prebere staršev `__index` in sledi referenci v tem [13]. Če poskusimo dodeliti vrednost ključu, ki v danem elementu še ne obstaja, se ta ključ ustvari in dodeli se mu vrednost – pri dodeljevanju dedovanja ni in če starševski objekt vsebuje ta ključ, ostane njegova vrednost nespremenjena.

Večkratno dedovanje je nekoliko kompleksnejše. V polje `__index` danega objekta moramo vstaviti metodo, ki pozna vse starše in ji zaporedoma pregleduje, dokler ne najde željenega ključa. Ko je iskan ključ najden, ga metoda vrne in dani objekt ga lahko uporabi. Ta pristop je nekoliko počasnejši od preprostega enojnega dedovanja, saj zahteva iskanje po več objektih. Dodani zahtevnosti se je mogoče izogniti tako, da se metode iz staršev v otroke preprosto prekopira v celoti (podobno kot mešani objekti, opisani v poglavju o javascriptu), vendar pa s tem izgubimo prednosti hierarhije objektov – če se spremeni neko metodo v danem prototipu, bodo metode njegovih otrok ostale nespremenjene [13]. Na ta način bo porabljenega tudi nekoliko več prostora.

3.5 Omega

Projekt omega se je začel leta 1990 zaradi nezadovoljstva nad takratnimi objektno-orientiranimi jeziki ter kot poskus, da bi v statično tipiziran jezik vpeljali objektno-orientirane prototipe. Omega se je trudila biti jezik, ki je preprost v principih in enostaven za uporabo, splošen in uporaben na več področjih, varen ter karseda učinkovit v računskem smislu [5]. Podobno kot self je bila omega razvita predvsem v raziskovalne namene in danes več ni v uporabi, a je kljub temu vredna omembe, saj je poznana kot edini prototipni jezik, ki je večinoma uporabljal statično tipiziranje.

Z uporabo statičnega tipiziranja se je bila omega prisiljena odpovedati delegaciji kot načinu dedovanja in možnosti spreminjanja posameznih objektov, izpeljanih iz istega prototipa, lastnostim, ki obstajajo v večini objektno-orientiranih jezikih s prototipi. To jo še dodatno izpostavlja kot posebnost med njimi, zakaj točno je ta sprememba pristopa potrebna, je natančneje opisano pri primerjavi jezikov v poglavju 4.2.

3.6 Ostali jeziki

Kljub temu da objektno-orientirano programiranje s prototipi ni najpopularnejši pristop k programiranju, je bilo po tem konceptu razvitih precej več jezikov, kot smo jih omenili v tem poglavju. Dosti prototipnih jezikov je bilo razvitih v devetdesetih letih prejšnjega stoletja, ko so bili koncepti objektne-orientiranosti še sveži. Primer takega jezika je kevo, ki je bil razvit leta 1992 na Finskem. Inspiracija za njegov razvoj sta bila med drugim self in omega [17], bil pa je večkrat omenjan v člankih na temo objektno-orientiranega programiranja s prototipi kot primer jezika, ki je uporabljal konkatenacijo ter propagiranje za dedovanje [7]. Dandanes je razvoj keva žal zamrl in tudi na spletu je o njem napisanega bore malo, tudi njegova programska koda ni na voljo, ne v tekstovni, niti v prevedeni obliki.

Ker je objektno-orientirano programiranje s prototipi zanimiv pristop, so se nekateri odločili ustvariti jezike po tem pristopu tudi, da bi ga bolje razumeli in se ob tem kaj naučili. Primer takega jezika, ki je vsaj delno zaslovel, je io. Prva verzija jezika io je bila razvita leta 2002, jezik pa je bil od tedaj razširjen z mnogimi modernimi knjižnicami, med drugim Sockets za komunikacijo preko spleta ter OpenGL za tridimenzionalno grafiko. Zanimive lastnosti jezika io so sočasno izvajanje in asinhrona komunikacija med objekti, implementacija objektov z režami, podobnimi selfovim, ter podpora izjemam, ki se sprožijo ob napakah v programih. Žal je v zadnjih letih tudi razvoj jezika io počasi zastal, zaradi majhnega splošnega interesa za jezik pa ni pričakovati, da se bo to spremenilo.

Poglavje 4

Primerjava jezikov

To poglavje zajema natančnejšo primerjavo danih jezikov ter vsaj podobnosti med njimi, kjer so bile odkrite večje razlike ali zanimivejše posebnosti pa tudi te. V primerjavah so zapisane razlike v namenu jezikov, različne ideje avtorjev, različni pristopi k implementaciji raznih programskih struktur in podobno. Za boljši prikaz delovanja opisanih mehanizmov poglavje vsebuje tudi primere iz kode posameznih jezikov.

4.1 Izbira kriterijev

Kriteriji primerjav so bili večinoma izbrani zaradi svoje pomembnosti za določen koncept objektno-orientiranega programiranja, ker so v objektno-orientiranih jezikih z razredi ter objektno-orientiranih jezikih s prototipi implementirani zelo različno ali pa, ker so na splošno pomembni pri izbiri programskega jezika za določeno nalogo.

Dedovanje je samo po sebi umeščeno med koncepte objektno-orientiranega programiranja. Poglobljena primerjava njegove implementacije med jeziki je tako nepogrešljiva v primerjalni nalogi, kot je vidno tudi v drugih podobnih študijah [5, 7, 8]. Različne vrste dedovanja so tudi ena od večjih razlik v

delovanju jezikov in prinašajo velike razlike v obnašanju podedovanih komponent ter načinu dela z objekti, ki jih vsebujejo.

Objekti so temelj objektno-orientiranega programiranja, uporabljani za praktično vse, a v raznih jezikih vseeno implementirani na zelo različne načine. Njihova stvaritev se posledično prav tako precej razlikuje od jezika do jezika, posebej zato, ker lahko nekateri jeziki objekte ustvarjajo tudi „ex nihilo“ [7], torej iz ničesar, medtem ko jih drugi lahko le izpeljujejo iz že obstoječih objektov preko dedovanja. Tako je primerjava ustvarjanja novih objektov dovolj pomembna [2, 5] in zanimiva, da si zasluži nekaj pozornosti.

Nadzor dostopa do komponent posameznih objektov je koncept, ki je tesno povezan s splošno idejo enkapsulacije v objektno-orientiranih jezikih, ta pa je pogosto natančneje obravnavana v literaturi [2, 5]. Enkapsulacija narekuje, da objekt navzven kaže le tiste komponente, ki jih želi in da sam nadzoruje, na kakšen način dovoljuje dostopanje do njih. Tega koncepta se nekateri jeziki držijo bolje kot drugi, ki ga ne vidijo kot zelo pomembnega in ga v veliki meri ignorirajo. Z natančnejšo primerjavo lahko tako ugotovimo, kateri jezik izbrati, če nam je natančen nadzor dostopa do posameznih komponent objektov pomemben.

Prenosljivost je pomembna, ker lahko jezik, ki deluje na več platformah, pokrije večje število uporabnikov in njihovih potreb kot tudi načinov uporabe. To daje jeziku večji doseg, programom, napisanim v njem, pa potencialno širšo ciljno publiko. Z izbiro jezika, ki podpira več različnih platform, lahko tudi znižamo cene razvoja aplikacij, saj nam za podporo različnih platform ni potrebno uporabljati različnih tehnologij.

Hitrost delovanja in splošna efektivnost implementacij jezikov sta pomembni iz več razlogov. V uporabniških aplikacijah hitrost delovanja zvišuje odzivnost in izboljšuje uporabniško izkušnjo, pri sistemskih opravilih pohi-

tri delovanje celotnega sistema, pri raziskovalnih namenih omogoča hitrejši dostop do rezultatov itd. Odvisna je od množice spremenljivk, vse od moči procesorja ter hitrosti pomnilnika do kompleksnosti idej v jeziku ter optimizacij v prevajalniku. Dobra implementacija jezika lahko delovanje programov pohitri za več kot desetkrat, ne le med različnimi jeziki, temveč tudi znotraj istega. Da bi vsaj približno pokazali primerjavo hitrosti delovanja različnih obravnavanih jezikov, smo smo del poglavja namenili tudi temu.

4.2 Dedovanje

Dedovanje je ena izmed osnovnih lastnosti ter prednosti objektno-orientiranega programiranja, saj omogoča uporabo istih funkcij in podatkov v različnih objektih, večinoma brez podvajanja kode. Obstaja več pristopov k izvajanju dedovanja in tudi jeziki s podobnimi pristopi se pogosto razlikujejo v svojih posebnostih.

V naši primerjavi se bomo osredotočili na dva glavna pristopa: klasično nasledstvo (angl. inheritance) ter delegacijo (angl. delegation). Delegacija je postopek, ki je značilnejši za objektno-orientirane jezike s prototipi, vendar pa je od obeh tudi računsko zahtevnejši. Klasični objektno-orientirani jeziki z razredi v glavnem uporabljajo nasledstvo.

Prednost klasičnega nasledstva je, da se povezave na podedovane komponente ustvarijo že pri prevajanju programa in tako povpraševanje po njih med izvajanjem ni potrebno. Ta postopek zahteva, da se že ob prevajanju ve, kje se nahajajo določene komponente, kakšnega tipa so ter v primeru metod še, kakšne argumente potrebujejo. Iz tega razloga potrebuje nasledstvo za svoje delovanje statično tipizirane komponente, katerih tipi se ne morejo poljubno spreminjati. Pomembno je tudi, da strukture samih objektov ostanejo takšne, kot so. Če se objekte lahko po začetni stvaritvi posamično spremi-

nja, lahko pride do situacije, kjer (denimo) dedovana metoda nepričakovano izgine ali spremenljivka zamenja svoj tip in nastane problem. Statično tipiziranje ter posamezno spremenljivi objekti tako ne morejo soobstajati [5].

Pri dinamično tipiziranih jezikih, v katerih se posamezen „primerek“ objekta lahko poljubno spreminja, pride tako v poštev delegacija. Ta je manj omejujoča glede sestave objektov: ni pomembno, da je v vsakem objektu zapisano, katere komponente je podedoval. Če se pojavi zahteva po komponenti, ki je objekt ne pozna, se zahteva preprosto posreduje njegovim prednikom in postopek se ponovi. S tem seveda ni nujno zagotovljeno, da neka komponenta sploh obstaja in programerjeva dolžnost je, da poskrbi, da komponente prototipov ostanejo takšne, kot jih potrebujejo njihovi izpeljani objekti.

O dedovanju v posameznih jezikih je bilo precej povedanega že v njihovih predstavitvah, z izjemo omege pa se precej dobro držijo delitve na nasledstvo in delegacijo. Java, kot objektno-orientiran jezik z razredi, uporablja enojno nasledstvo. Self, lua in javascript uporabljajo delegacijo, pri čemer je self najmanj omejujoč in omogoča celo dinamično dodajanje in odstranjevanje nasledstvenih povezav med izvajanjem programa, kar imenuje dinamično (angl. dynamic), večkratno dedovanje. Lua podpira tako enojno kot tudi večkratno dedovanje z delegacijo, slednjo z nekoliko počasnejšim izvajanjem. Javascript podpira le enojno dedovanje preko delegacije.

Izjema v tej primerjavi je omega, kar je tudi razlog za njeno omembo. Zaradi svojega statičnega tipiziranja ne more uporabljati delegacije, zato implementira bolj klasičen pristop z uporabo nasledstva. Zato v njej tudi ni dovoljeno spreminjanje posameznih objektov. Ob spremembi prototipa se ista sprememba propagira navzdol po objektih, izpeljanih iz njega. Če želimo iz enega prototipa izpeljati nov prototip, je to mogoče, priredimo pa ga lahko na sledeče načine: dodajamo lahko nove spremenljivke in metode, podedo-

vane metode lahko povozimo (angl. override), možno pa je tudi spreminjati vsebino novo dodanih spremenljivk. Zadnja lastnost omego tudi loči od razrednih objektno-orientiranih jezikov, v katerih to ni mogoče in je potrebno vsebino spremenljivk dokončno določiti pri iniacializaciji objektov [5].

jezik	**način dedovanja**
java	klasično nasledstvo
self	delegacija, večkratno dinamično dedovanje
javascript	delegacija, enkratno dedovanje
lua	delegacija, večkratno dedovanje
omega	klasično nasledstvo

Tabela 4.1: Pregled dedovanja

V Tabeli 4.1 so na kratko povzeti načini dedovanja v obravnavanih jezikih, sedaj pa se bomo posvetili še bolj praktični primerjavi dedovanja ter dela z objekti v obeh tipih jezikov.

V programu 4.1 vidimo primer postavitve razredov za prikaz dedovanja v javi. Razred `Zival` ima dva naslednika – `Pes` in `Macka`. Ime živali se ob stvaritvi objekta zabeleži, kar je opisano v konstruktorju razreda `Zival`. Konstruktorja obeh podrazredov lahko tega pokličeta s klicem `super()` in tako zabeležita svoji imeni. Tako `Pes` kot `Macka` od razreda `Zival` podedujeta funkcijo `izpisiIme()`, ki izpiše ime živali, vsak pa tudi definira svoje oglašanje v funkciji `lajaj()` (za psa) ter `mjavkaj` (za mačko). Živali so tako zastavljene, kakor so, pes bo vedno pes in mačka vedno mačka, oba pa lahko uporabljata enako funkcijo za izpis svojega imena. V programu 4.2 vidimo, kako iz prej definiranih razredov naredimo primerke objektov `Pes` in `Macka` ter pokličemo funkcije za izpis imena in oglašanje.

V prototipnih jezikih podoben primer ni nujno tako dokončen. V programu 4.3 v javascriptu ustvarimo objekt `zival` in mu dodelimo funkciji,

s katerima lahko nastavimo in izpišemo ime živali. Ustvarimo objekta `rex` in `mimi`, ki oba uporabljata objekt `zival` kot prototip. Rexu, ki je pes, napišemo funkcijo `lajaj()`, ki v konzolo izpiše „Hov!“ Mimi, ki je mačka, dobi funkcijo `mijavkaj()`, ki izpiše „Mjav!“ Rexu nastavimo njegovo ime s klicem funkcije iz objekta `zival`, nato storimo isto še za Mimi. Obe živali imamo sedaj predstavljeni s svojimi objekti. V programu 4.4 pokažemo, kako objekta delujeta. S klicem `izpisiIme()` oba objekta uporabita metodo, podedovano iz objekta `zival` in izpišeta ime živali. Ko Rexu ukažemo `lajaj()`, izpiše „Hov!“ in Mici izpiše „Mjav!“ na ukaz `mjavkaj()`. Nato se zgodi čudež – Mimi se spremeni v psa. Objektu `mimi` tako odstranimo funkcijo `mjavkaj()` (psi ne mjavkajo), ter dodelimo funkcijo `lajaj()`, kot jo uporablja že objekt `rex`. Klic `mimi.lajaj()` nam pove, da Mimi sedaj laja, kot vsi pravi psi. Obe živali nato še naučimo, kako se da tačko z definicijo funkcije `zival.dajTacko()`, ter preizkusimo, da to sedaj resnično znata.

Tako spreminjanje objektov v objektno-orientiranih jezikih z razredi (konkretno javi) načeloma ni mogoče, saj morajo razredi vsebovati dokončne opise objektov, ki se potem, ko jih instanciramo, ne morejo več spreminjati. V prototipnih jezikih lahko objekt najprej ustvarimo iz prototipa, nato pa mu poljubno dodajamo in odvzemamo komponente. Tudi če se sredi programa odločimo, da prototip potrebuje dodatno komponento, mu jo lahko dodamo. Takoj po tem bodo do nove komponente lahko dostopali tudi objekti, ki so iz prototipa izpeljani, kar demonstriramo s funkcijo `dajTacko`. Klasično nasledstvo objektno-orientiranih jezikov, ki uporabljajo razrede, je sicer bolj optimizirano in deluje hitreje, vendar pa nam prototipni pristopi (kot je delegacija) pogosto omogočajo večjo dinamičnost objektov ter lažje delo z njihovimi komponentami.

Program 4.1: Nastavek za dedovanje v javi

```
class Zival {
    private String ime;
    public Zival(String ime) {
```

```
        this.ime = ime;
    }
    public void izpisiIme() {
        System.out.println(ime);
    }
}

class Pes extends Zival {
    public Pes(String ime) {
        super(ime);
    }
    public void lajaj() {
        System.out.println("Hov!");
    }
}

class Macka extends Zival {
    public Macka(String ime) {
        super(ime);
    }
    public void mjavkaj() {
        System.out.println("Mjav!");
    }
}
```

Program 4.2: Izvajanje dedovanja v javi

```
public class Dedovanje {
    public static void main(String[] args) {
        Pes rex = new Pes("Rex");
        Macka mici = new Macka("Mici");

        rex.izpisiIme();     // Rex
        rex.lajaj();         // Hov!

        mici.izpisiIme();    // Mici
        mici.mjavkaj();      // Mijav!
    }
}
```

Program 4.3: Nastavek za dedovanje v javascriptu

```
var zival = Object.create(Object);
```

```
zival.nastaviIme = function(ime) {
    this.ime = ime;
}
zival.izpisiIme = function() {
    console.log(this.ime);
}

var rex = Object.create(zival);
var mimi = Object.create(zival);

rex.lajaj = function() {
    console.log("Hov!");
}
mimi.mjavkaj = function() {
    console.log("Mjav!");
}
rex.nastaviIme("Rex");
mimi.nastaviIme("Mimi");
```

Program 4.4: Delo z objekti v javascriptu

```
rex.izpisiIme();        // Rex
rex.lajaj();            // Hov!
mimi.izpisiIme();       // Mimi
mimi.mjavkaj();         // Mjav!

delete mimi.mijavkaj;   // Mimi naenkrat ne zna vec mijavkati.
mimi.lajaj = rex.lajaj; // Mimi se cudezno spremeni v psa.
mimi.lajaj();           // Hov!

// Obe zivali naucimo dati tacko, spreminjamo prototip.
zival.dajTacko = function() {
    console.log("tacka");
}
rex.dajTacko();         // tacka
mimi.dajTacko();        // tacka
```

4.3 Ustvarjanje objektov

Objekti so glavne komponente objektno-orientiranih jezikov, v katerih se uporabljajo praktično povsod, vendar pa jih moramo, če jih želimo uporabljati,

najprej nekako ustvariti.

V razrednih objektno-orientiranih jezikih poteka ustvarjanje objektov preko razredov. V razred se zapišejo definicija objekta, njegovih komponent in obnašanja, nato pa se naredi primerek tega razreda in dobimo objekt, pripravljen za uporabo. V objektno-orientiranih jezikih s prototipi se do objektov načeloma pride s kloniranjem: vzamemo obstoječ objekt (prototip), naredimo kopijo (ki je bolj ali manj odvisna od originala ter njegovih komponent) in dobimo nov objekt, ki je pripravljen na spreminjanje ali pa kar na neposredno uporabo.

Seveda pa ni rečeno, da bomo pri kreaciji novega objekta vedno potrebovali komponente, ki smo jih že uporabili drugje. Če potrebujemo preprost objekt, ki bo hranil dve števili in en znak, navadno ni potrebno, da podeduje celotno knjižnico za delo z besedilom. Za primere, ko pri ustvarjanju objekta ne želimo uporabiti prototipa, ponujajo nekateri objektno-orientirani jeziki s prototipi kreacijo „ex-nihilo“, torej iz ničesar. Objekti, ustvarjeni na ta način, so sveže stvaritve, ki ne podedujejo ničesar od nikoder.

Javascript omogoča stvarjenje objektov iz ničesar tako, da se za prototip nastavi vrednost `null`, torej prazno vrednost. Potrebno je biti previden, saj bo javascript za prototip novim objektom avtomatsko nastavil osnovni objekt `Object`, če mu željenega prototipa (ali vrednosti `null`) ne podamo eksplicitno. Najlažji način za stvaritev svežega, praznega objekta v javascriptu je tako klic `Object.create(null)` [10], ki nam tudi javi napako, če prototipa ne nastavimo eksplicitno. V lui je kreacija objektov ex-nihilo prav tako mogoča, potrebno je le, da polje `__index` novega objekta pustimo prazno. Tudi self z ex-nihilo objekti nima problemov – prototipe je objektom potrebno določiti ročno; če tega ne storimo objekti ne podedujejo ničesar. Kot izjema se spet pojavi omega, ki kreacije objektov ex-nihilo ne podpira [7].

V Tabeli 4.2 lahko najdemo pregled možnosti ustvarjanja objektov v obravnavanih jezikih, dejanske primere ustvarjanja objektov pa smo pokazali v programih 4.1 in 4.2 za javo ter 4.3 za javascript.

jezik	**ustvarjanje objektov**
java	instanciranje razredov
self	prototipno kloniranje, ex-nihilo
javascript	prototipno kloniranje, ex-nihilo
lua	prototipno kloniranje, ex-nihilo
omega	prototipno kloniranje

Tabela 4.2: Pregled ustvarjanja objektov

4.4 Nadzor dostopa

Zasebnost oziroma nadzor dostopa je mehanizem v nekaterih jezikih, ki nadzoruje, od kod je mogoče dostopati do komponent določenega objekta. Čeprav tega pristopa vsi jeziki ne implementirajo eksplicitno ali pa ga sploh ne, je vseeno uporaben za večjo varnost jezika, preprečevanje napačne rabe spremenljivk (na primer preprečevanje neposrednega branja neke komponente objekta, ko je za dostop do njene vrednosti na voljo posebna metoda, ki vrača drugačen rezultat), ter upoštevanje koncepta enkapsulacije.

Naš razredni jezik, java, ponuja tri možne, izrecno zapisane nivoje dostopa do posamezne komponente: zasebni dostop, pri katerem je mogoče do komponente dostopati le znotraj razreda, zaščiteni, ki omogoča dostop do komponente iz razredov istega paketa, in podrazredov, ki komponento podedujejo, ter javni, ki omogoča dostop do komponente od koderkoli. Če ob najavi komponente dostopa do nje ne omejimo izrecno, se zanjo nastavi privzet dostop, ki omogoča dostop do nje iz istega razreda ter razredov v istem

paketu [18].

Lua sama po sebi ne ponuja možnosti za nadzor dostopa – vse je javno in na voljo od koderkoli. Razlog za to se delno skriva v njeni preprosti implementaciji objektov s tabelami, delno pa tudi v sami filozofiji jezika, ki v osnovi ni načrtovan za uporabo v velikih projektih z mnogimi različnimi razvijalci (čeprav se dandanes z uporabo v razvoju iger to nekoliko spreminja), pač pa je bolje prilagojen manjšim in srednje velikim programom, ki se lahko nato morda vključijo v večji projekt. Vseeno je v lui vsaj zaseben dostop mogoče implementirati, kar je demonstrirano v programu 4.5. Objekt za ta primer ustvarimo s pomočjo funkcije `ustvariPrivatno`, ki je tako podobna konstruktorskim metodam, značilnim za jezike, ki uporabljajo razrede. V tej funkciji implementiramo metode za dostop do zasebne spremenljivke `vrednost`, ki jo shranimo v lokalni objekt `privatna`. Ker objekt `privatna` obstaja le znotraj te funkcije, do njega ne bo več mogoče dostopati preko tega imena, potem ko se bo funkcija končala. Lahko pa napišemo metode za delo s tem objektom (v našem primeru `pisi` in `beri`), ki jih nato vrnemo v novem objektu kot rezultat funkcije. Vrnjeni objekt tako ne vsebuje nobene neposredne reference na zasebno spremenljivko `vrednost`, lahko pa do nje dostopamo preko njegovih metod `pisi` in `beri`.

Program 4.5: Zasebni dostop v lui

```
-- funkcija, shranjena v datoteko "privacy.lua"
function ustvariPrivatno (zacetna)
    local privatna = {vrednost = zacetna}

    local pisi = function (nova)
        privatna.vrednost = nova
    end

    local beri = function ()
        return privatna.vrednost
    end

    return {
        pisi = pisi,
        beri = beri
```

```
    }
end

-- v ukazni vrstici
> dofile("privacy.lua")
> privatna = ustvariPrivatno(5)
> privatna.beri()    --beri zasebno vrednost
5                    --dobimo rezultat 5
> privatna.pisi(8)   --pisi zasebno vrednost
> privatna.beri()    --beri zasebno vrednost
8                    --dobimo rezultat 8
> privatna.privatna --poskus direktnega dostopa
nil                  --dobimo prazno vrednost
> privatna.vrednost --poskus direktnega dostopa
nil                  --dobimo prazno vrednost
```

Tudi javascript nima namenskih ukazov za omejevanje dostopa in vse komponente objektov obravnava kot javne, vendar pa je podobno kot pri lui v njem mogoče ročno implementirati zaseben dostop do nekaterih spremenljivk. Zasebno spremenljivko je mogoče ustvariti na zelo podoben način kot v lui – v konstruktorju objekta jo najavimo kot lokalno spremenljivko z uporabo besede `var`, nato pa (prav tako v konstruktorju) ustvarimo spremenljivke z anonimnimi funkcijami za dostop do nje. Ker je spremenljivka lokalna in uporabljena (in torej vidna) le znotraj anonimnih funkcij, je dostop do nje mogoč le preko njih [9]. Javascript od specifikacije ecmascript 5 (v večini brskalnikov celo že od ecmascript 3) podpira tudi funkciji za pridobivanje (angl. getter) ter nastavljanje (angl. setter) vrednosti [10]. Ti sta pogosto asociirani z zasebnimi spremenljivkami ter služita za omejen dostop do njih (na primer v javi), vendar pa v tem primeru ne opravljata dejanske kontrole dostopa, temveč le ponujata možnost izvajanja dodatnih operacij, ko se do spremenljivk dostopa preko njiju. Dostop do spremenljivk znotraj njiju je namreč še vedno mogoč neposredno z uporabo njihovih imen.

V selfu nadzor dostopa sicer obstaja, vendar pa je le navidezen – za različne nivoje dostopa se v uporabniškem vmesniku imena rež prikazujejo z različnim slogom pisave, vendar pa self sam za dejansko omejevanje dostopa

do teh rež ne skrbi. Ta mehanizem služi predvsem jasnosti programa in omogoča programerju, da nakaže, kaj je namen določene reže oziroma kako naj bi se jo uporabljalo. Self lahko tako nakaže tri vrste predvidenega dostopa: zasebnega, javnega ter nedefiniranega, ki je edinstven v selfu, pomeni pa le, da ni točno predvideno, od kod naj bi bila reža vidna.

jezik	**nadzor dostopa**
java	da, zasebno / zaščiteno / javno / privzeto
self	ne, samo navidezen kot komentar
javascript	delno, ni izrecen
lua	delno, ni izrecen

Tabela 4.3: Pregled nadzorov dostopa

Tabela 4.3 vsebuje kratek pregled nadzorov dostopa v izbranih programskih jezikih.

4.5 Prenosljivost

Prenosljivost je lastnost programskih jezikov in programov, ki nam omogoča, da jih poganjamo na več različnih platformah, z različnimi operacijskimi sistemi, procesorskimi arhitekturami ter ostalimi strojnimi komponentami. Večja prenosljivost pomeni, da isti program lahko deluje na več platformah ter s tem viša uporabnost jezika ter niža ceno razvoja. Objektno-orientirani jeziki so na splošno precej visoko nivojski, kar pomeni, da pri komunikaciji s strojno opremo ponavadi uporabljajo več abstrakcije in so zato bolj prenosljivi.

Java je znana kot eden najbolj razširjenih jezikov na svetu, že dolga leta pa se jo uporablja na mnogih področjih. Pogosto je uporabljena za programiranje večjih poslovnih aplikacij, spletnih strežnikov ter namiznih aplikacij,

kot glavni programski jezik jo najdemo v večini modernih pametnih telefonov, prirejeno različico, java platforma, mikro različica (angl. Java Platform, Micro Edition), pa tudi v starejših telefonih in enostavnejših brezžičnih napravah (internet stvari). Osnovna lastniška različica jave podpira operacijske sisteme Linux (x86, x64), Mac OS X (x64), Solaris (x86, x64, SPARC) in Windows (x86, x64) [19].

Podpora za javascript je zelo široka. Podpirajo ga vse platforme z dostopom do spleta in vsaj približno sodobnim brskalnikom, saj spada med osnovne spletne tehnologije in je danes za pravilno delovanje mnogih spletnih strani praktično nepogrešljiv. Implementacij javascripta je sicer veliko, a ker obstaja enoten standard zanje (ecmascript), so te med seboj bolj ali manj kompatibilne.

Lua je v svoji osnovni implementaciji knjižnica, zapisana v jeziku C, z dodatkom malega tolmača (angl. interpreter), ki s knjižnico komunicira. Lua tako deluje na praktično vseh sistemih, na katerih deluje programski jezik C. Ker je C poleg raznih zbirnih jezikov osnova za večino operacijskih sistemov, deluje lua praktično povsod. Da se prilagodi preprostejšim napravam, recimo vgrajenim sistemom, omogoča več nastavitev glede dostopa do spomina ter odstranjevanja delov knjižnice ob prevajanju. Na ta način jo je mogoče spraviti tudi na naprave s preprostejšimi strojnimi komponentami ter malo razpoložljivega spomina [13].

Za razliko od lue luaJIT ni tako prenosljiv. Glavni razlog za to je, da je luaJIT sam po sebi prevajalnik, kar pomeni, da mora zgenerirati strojno kodo [20], medtem ko se lahko običajna implementacija lue s to nalogo obrne na C. Kljub temu da luaJIT ne podpira vseh platform, ki jih podpira C, jo je vseeno mogoče uporabljati na veliko različnih sistemih – Linux, OS X, Windows ter BSD in podobne Unix sisteme na namizju, Android in iOS na mobilnih napravah, več konzol PlayStation ter Xbox itd. Od procesorskih

arhitektur podpira x86, x64, ARM, PowerPC ter MIPS [20].

Self je od omenjenih, še živih jezikov najmanj prenosljiv. Glavni razlog za to je verjetno, da prenosljivost ni bila njegov cilj. Jezik self je bil napisan bolj za raziskovanje novih konceptov prototipiranja ter objektno-orientiranega programiranja na sploh, kot za pisanje zmogljivih, prenosnih programov. Dandanes tako deluje na Linuxu ter Mac OS X, starejše različice pa podpirajo še Solaris [1].

V Tabeli 4.4 najdemo pregled podpore obravnavanih jezikov za različne operacijske sisteme ter platforme.

jezik	**podprti sistemi**
java	Linux, OS X, Solaris, Windows + Micro Edition za razne manjše / brezžične naprave
self	Linux, OS X
javascript	odvisno od brskalnikov, zelo razširjen
lua	vsi glavni – vse, kar podpira C
luaJIT	Linux, OS X, Windows, BSD in podobni, Android, iOS; konzole PlayStation, Xbox; arhitekture x86, x64, ARM, PowerPC, MIPS

Tabela 4.4: Podpora jezikov za platforme

4.6 Primerjava hitrosti izvajanja

V tem razdelku se nekoliko oddaljimo od teoretične podlage ter pristopov v posameznih jezikih in se bolj osredotočimo na njihovo uporabo, natančneje na njihovo hitrost delovanja. Za nalogo je bilo izbranih ter sestavljenih nekaj algoritmov, ki izvajajo operacije, pogoste pri delu z jeziki: kreacijo objektov,

spreminjanje njihovih lastnosti, klicanje njihovih metod ter podedovanih metod njihovih predhodnikov, rekurzija ipd.

Kar preverjamo hitrost delovanja jezikov in s tem del njihove uporabne vrednosti, je smiselno, da v tem razdelku uporabimo jezike, ki so še živi (torej aktivno v razvoju), široko uporabljani, dovolj optimizirani ter dejansko v široki uporabi. Self v tem primeru odpade, saj gre za bolj akademski jezik, in čeprav se počasi usmerja tudi v „resno programiranje“ [1], ni uporabljen v nobenem (avtorju poznanem) večjem projektu. Tudi omega za to poglavje ni primerna, saj je jezik dandanes bolj teoretičen kot dejansko uporaben, več ni v aktivnem razvoju, na voljo pa tudi ni nobene implementacije za moderne platforme.

Jeziki, ki nam torej ostanejo za testiranje, so java, javascript ter lua. Testi so izvedeni s 64-bitnimi različicami jezikov, posamezne uporabljene implementacije pa so:

- java - OpenJDK 1.8.0_91
- javascript - brskalnik Chromium 51 z javascript pogonom V8, verzija 5.1.281.57
- lua 5.3.1
- luaJIT 2.0.4

Ker ima vsak od jezikov več različnih implementacij, je potrebno pojasnilo na temo izbire samih implementacij. Java ima dve osnovni verziji – uradno lastniško različico ter odprtokodni OpenJDK. OpenJDK zadnje čase vedno bolj prihaja v ospredje kot glavna verzija jave, in ker med njima ni več ogromnih razlik, je bil izbran OpenJDK. Javascript ima več implementacij, saj ga različni brskalniki izvajajo različno. Chromium (oziroma njegov skoraj identični dvojček Chrome) je na splošno poznan kot eden hitrejših brskalnikov, je pa trenutno tudi daleč najpopularnejši in tako smiseln kandidat za

testiranje javascripta. Njegov skriptni motor, V8, prevaja javascript naravnost v strojno kodo in s tem omogoča hitro delovanje [11]. Pri lui sta izbrani dve implementaciji, ki se med seboj po delovanju precej razlikujeta. Navadna lua je knjižnica za programski jezik C, v njem napisana ter izvajana kot skriptni jezik. LuaJIT je implementacija JIT prevajalnika za luo, ki njeno delovanje še dodatno pohitri s prevajanjem delov programske kode naravnost v strojno kodo – podobno, kot to počne java. Ker je navadna lua med obema popularnejša in ker je uradna verzija jezika, jo je seveda smiselno vključiti, ker pa je tudi luaJIT precej široko uporabljana različica, katere celoten smisel je, da pohitri delovanje jezika, je tudi testiranje te zanimivo.

4.6.1 Postavitev testov

Hitrost izvajanja je merjena v milisekundah realnega časa (torej ne časa, ki ga za izvajanje programa porabi centralna procesna enota). Ker imajo jeziki različne pristope k merjenju časa, med katerimi so nekateri rahlo kompleksnejši od drugih, je možno, da bo ponekod sam klic za meritev porabil več časa kot drugje ter tako vplival na rezultat. Da bi zmanjšali vpliv te ter podobnih neželenih spremenljivk, bomo uporabljali nekoliko večje testne primere. Testi so bili izvedeni na operacijskem sistemu Ubuntu 16.04 z Linux jedrom 4.4.0, na računalniku s procesorjem Intel Core i7-3630QM (4 fizična jedra, 8 navideznih) in 8 GB delovnega pomnilnika.

V javi je merjenje časa preprosto, ukaz `System.nanoTime()/100000` pridobi čas od zagona javanskega navideznega stroja v nanosekundah, nato pa ga z deljenjem pretvori v milisekunde. V javascriptu je situacija podobna, milisekunde v Unix času (od polnoči na 1. januar 1970 po UTC času) se dobi z ukazom `Date.now()`.

Lua na Linuxu ne daje na voljo klica, ki bi preprosto vrnil realen čas v milisekundah ali manjših enotah, zato se s to nalogo obrnemo na sistemsko ukazno lupino Bash, na način, prikazan v programu 4.6. Ta pristop de-

luje tako v navadni lui kot v luaJIT, porabi pa okoli eno milisekundo časa, nekoliko manj v luaJIT kot v lui. Funkcija najprej izvede Bash ukaz za pridobivanje časa v nanosekundah in shrani rezultat ukaza v datoteko. Drugi ukaz datoteko prebere in shrani rezultat v spremenljivko, tretji pa vrednost pretvori iz nanosekund v milisekunde in rezultat vrne.

Program 4.6: Merjenje časa v lui

```
gettime = function ()
    timerfile = io.popen("date +%s%N")
    timerval = timerfile:read()
    return math.floor(timerval / 1000000)
end
```

Poleg časovnih rezultatov izvajanja so v teste vključeni tudi okvirni grafi obremenjenosti jeder centralne procesne enote med izvajanjem testov. Njihov primaren namen je demonstracija obtremenjenosti procesorja pri izvajanju testa ter trajanje in intenzivnost te obremenitve. Grafi so izdelani z orodjem Gnome System Monitor [21]. Abscisna os (x) predstavlja trajanje, pri čemer pomeni en razdelek deset sekund, ordinatna os (y) pa intenzivnost obremenitve, kjer pomeni en razdelek dvajset odstotkov skupne zmogljivosti procesorskega jedra.

Za vsak test je bilo v vsaki od testiranih implementacij narejenih dvanajst meritev, od katerih sta bili dve najdlje od povprečja zavrženi. Iz ostalih desetih meritev so bile izračunane sledeče statistike:

- aritmetična sredina (AS), ki predstavlja povprečno vrednost meritev ter s tem povprečni čas izvajanja,
- standardni odklon / standardna deviacija (SD), ki predstavlja razpršenost rezultatov, torej velikost razlik med njimi,

- koeficient variacije (KV), ki je razmerje med standardnim odklonom ter aritmetično sredino in z odstotki odstopanja bolje ilustrira razpršenost rezultatov.

Poleg zapisanih kriterijev vsebujejo tabele primerjav še polje "razmerje", ki je normirano na čas izvajanja v najhitrejšem od jezikov oziroma implementacij. Za vsak jezik pove, kakšno je razmerje njegovega časa izvajanja testa proti najhitrejšemu. Glavni namen tega polja je, da da bralcu hitro predstavo o razlikah rezultatov.

4.6.2 Učinkovitost funkcijskih klicev

Test merjenja hitrosti izvajanja funkcijskih klicev je namenjen preizkušanju učinkovitosti (rekurzivnih) klicev funkcij v vsakem od jezikov. Ker funkcija kliče samo sebe, se klicoča in klicana funkcija nahajata znotraj istega objekta; velikih idejnih razlik pri implementaciji takih klicev med razrednimi in prototipnimi objektno-orientiranimi jeziki tukaj ni. Gre za splošen test učinkovitosti implementacije klicev funkcij.

Ta test je nekoliko zahtevnejši od ostalih, kar je vidno tudi iz rezultatov. Sama koda testa je rekurzivno računanje n-tega Fibonaccijevega števila (v izvedenih testih štiridesetega), izrezki njene implementacije v različnih jezikih pa se nahajajo v Dodatku A, v primerih programov A.1 za javo, A.2 za javascript ter A.3 za luo. Ker vsak klic funkcije kliče isto funkcijo še dvakrat, je zahtevnost programov $\Omega(\varphi^n)$, kar nam pri `n = 40` prinese nekaj čez bilijon klicev.

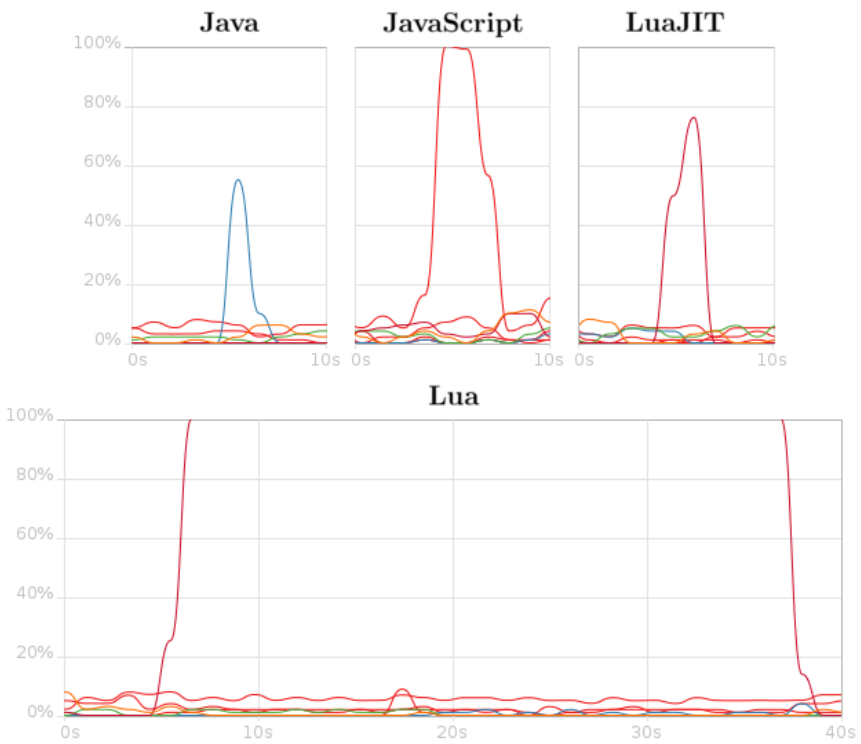
Kot je razvidno iz tabele 4.5, izvede java test najhitreje. Javascript potrebuje za izvajanje primera več kot štirikrat toliko dolgo, lua pa celo slabih tridesetkrat, kar jo postavlja na zadnje mesto. Druga implementacija lue, luaJIT, se odreže precej bolje in porabi le dobrih dvakrat toliko časa kot java. Ker gre v primeru programa za veliko zaporednih klicev malega odseka kode,

lahko sklepamo, da jo tako javin kot luaJITov JIT prevajalnik uspešno zaznata kot ponavljajočo in jo prevedeta. Tako pohitrita izvajanje ter dosežeta dosti boljše rezultate.

Slika 4.1 prikazuje obremenjenost procesorja, pri izvajanju tega testa. Pri javi ter luaJIT obremenitev jedra sploh ne doseže stotih odstotkov, pri javascriptu le za kratek čas, nato pa upade, pri lui pa jedro ostane polno obremenjeno skoraj cele pol minute izvajanja testa.

jezik	**AS [ms]**	**razmerje**	**SD [ms]**	**KV [%]**
java	565,00	1,00	4,76	0,84
javascript	2630,00	4,65	10,71	0,41
lua	29736,00	52,63	203,98	0,69
luaJIT	1271,80	2,25	6,16	0,48

Tabela 4.5: Rezultati testiranja učinkovitosti funkcijskih klicev



Slika 4.1: Obremenjenost procesorja pri testih učinkovitosti funkcijskih klicev

4.6.3 Dedovanje

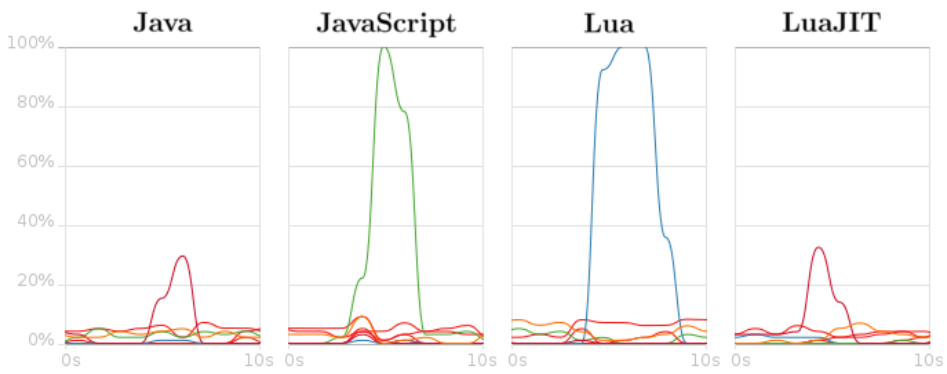
V testu dedovanja se osredotočimo na delovanje klicev metod, ki jih je dani objekt podedoval. V testu generiramo veliko tabelo naključnih števil od 0 do 3 (oziroma 1 do 4 v primeru lue), nato pa se metoda v objektu sprehodi po njej ter glede na število v tabeli pokliče metodo. Vse metode pripadajo „staršem“ objekta, ki jih kliče, starši pa so postavljeni v verigo, tako da je vsaka od metod na svoji globini dedovanja. Objekt tako dostopa do metod na naključnih globinah ter jih uporablja za preproste izračune, mi pa merimo čas, ki ga za to porabi. Implementacije klicne funkcije, vključno s kreacijo objektov, se nahajajo v Dodatku A, v primerih kode A.4 za javo, A.5 za javascript ter A.6 za luo. Testiranje se je izvajalo z 10^7 klicev.

Po meritvah časa, vidnih v Tabeli 4.1, vidimo, da se java pričakovano odreže najbolje. Zaradi bolj statične narave klasičnega nasledstva se povezave do podedovanih komponent ustvarijo že pri prevajanju programa in tako se med izvajanjem ne porablja časa z iskanjem želene komponente po verigi dedovanja. Kljub temu je bil v tem testu javascript le za slabo petino počasnejši. LuaJIT je porabil skoraj štirikrat toliko časa kot java, običajna lua pa kar tridesetkrat.

Prikazi obremenjenosti procesorja na sliki 4.2 pri tem primeru niso zelo natančni, saj velik del obremenjenosti pride iz generacije tabele naključnih števil. Merjenje časa se v vseh testih zgodi šele, ko je tabela generirana, tako da generacija nanj ne vpliva. Vseeno lahko vidimo, da java ter luaJIT procesor zaposlita občutno manj kot javascript in lua.

jezik	**AS [ms]**	**razmerje**	**SD [ms]**	**KV [%]**
java	68,50	1,00	0,53	0,77
javascript	81,20	1,19	0,63	0,78
lua	2110,00	30,80	11,99	0,57
luaJIT	265,00	3,87	2,00	0,75

Tabela 4.6: Rezultati testiranja hitrosti dedovanja



Slika 4.2: Obremenjenost procesorja pri testih dedovanja

4.6.4 Pošiljanje sporočil

V testu pošiljanja sporočil preverjamo, kako efektivno lahko objekti med seboj komunicirajo. Ker je pošiljanje sporočil en od temeljnih konceptov, na katerih sloni objektno-orientirano programiranje, je pomemno tako za razredne kot tudi za prototipne jezike. V našem testu objekt A ustvari število 0 ter ga poda metodi objekta B, da ga shrani. Objekt B število poviša za ena ter shrani. Nato pokliče metodo objekta A in ji poda shranjeno število, da ga prav tako poviša za ena ter shrani. Objekt A spet pošlje število (zdaj povečano) objektu B itd. Sporočila si pošiljata, dokler vrednost v objektu A ne doseže želene višine. Uporabljena koda se nahaja v Dodatku A, v primerih programov A.7 za javo, A.8 za javascript ter A.9 za luo. V našem testu je bilo izmenjav števil 10^7.

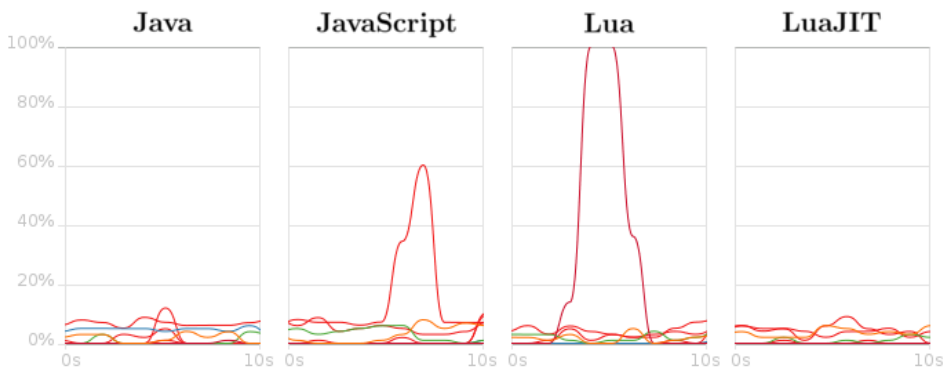
Pri kodi za pošiljanje sporočil pride do izraza programerska prednost prototipnih jezikov. Objekta A in B med sabo komunicirata in vsak od njiju pozna drugega kot svoj par (`this.pair`). Ker lahko objektom v prototipnih jezikih dodajamo komponente po tem, ko so že ustvarjeni, lahko objektoma A in B določimo drug drugega za par v dveh vrsticah po njuni stvaritvi. V

javi je za to operacijo potrebno obe komponenti najprej definirati v definicijah razredov, a ker jima tam vrednosti še ne moremo dodeliti (objekta še nista ustvarjena), je potreben še en dostop do vsakega od objektov po njegovi stvaritvi, da mu nastavimo drugi objekt kot par. Na sploh je pri prototipnem pristopu deklaracija razredov izpuščena in objekte se gradi sproti, kar je opazno enostavneje.

Kot lahko vidimo v Tabeli 4.7, se pri testiranju hitrosti izvajanja ponovno najbolje odreže java, ki test konča v slabih dvaintridesetih sekundah. Sledi ji luaJIT, ki porabi nekoliko več kot dvakrat toliko časa, javascript potrebuje sedemitridesetkrat toliko, lua pa je kar stodesetkrat počasnejša. Tudi pri primerjavi obremenjenosti procesorja, vidni na sliki 4.3, so grafi skladni z rezultati meritev hitrosti – java ter luaJIT procesorja skoraj ne obremenita, javascript ter lua pa mu dasta nekaj dela.

jezik	**AS [ms]**	**razmerje**	**SD [ms]**	**KV [%]**
java	22,80	1	0,92	4,03
javascript	849,30	37.25	7,23	0,85
lua	2515,70	110,34	48,42	1,92
luaJIT	53,5	2,35	4,28	7,99

Tabela 4.7: Rezultati testiranja hitrosti pošiljanja sporočil



Slika 4.3: Obremenjenost procesorja pri testih pošiljanja sporočil

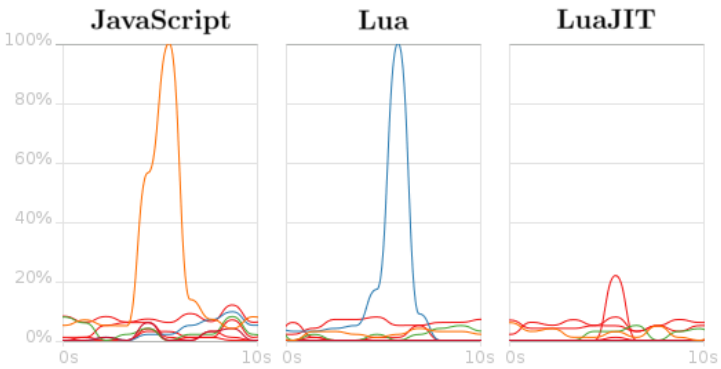
4.6.5 Dinamično dodeljevanje vrednosti

V testu dinamičnega dodeljevanja vrednosti preverimo, kako hitro lahko jezik isti spremenljivki dodeljuje vrednosti različnih tipov. Java v ta test ni vključena, saj je statično tipiziran jezik in takšnega menjavanja tipov spremenljivk ne omogoča. V testu v zanki menjamo vrednost spremenljivke iz prazne vrednosti (angl. null) v celo število, tekst, decimalno število, referenco na objekt ter nazaj v prazno vrednost. V testiranju je izvedenih 10^7 menjav tipa spremenljivke, koda v obeh jezikih pa je vključena v Dodatku A, v programu A.10 za javascript ter A.11 za luo.

Iz Tabele 4.8 z rezultati tega testa lahko razberemo, da se je v tem testu najbolje odrezal luaJIT, ki je potreboval 213 milisekund, sledi mu skoraj šestkrat počasnejša lua, javascript pa, skoraj osemkrat počasnejši, presenetljivo zasede zadnje mesto. Morda ima s temi rezultati kaj opraviti preprostost luinih objektov, ki so le rahlo prirejene asociativne tabele. Obremenjenost procesorja med testi, vidna na sliki 4.4, se ponovno ujema z rezultati testov hitrosti – procesor najbolj obremeni javascript, najmanj pa luaJIT.

jezik	AS [ms]	razmerje	SD [ms]	KV [%]
javascript	1679,60	7,86	39,08	2,33
lua	1223,20	5,73	8,56	0,70
luaJIT	213,60	1,00	2,59	1,21

Tabela 4.8: Rezultati testiranja hitrosti dinamičnega dodeljevanja vrednosti



Slika 4.4: Obremenjenost procesorja pri testih dinamičnega dodeljevanja vrednosti

4.6.6 Povzetek in razprava

V naših testiranjih se je pokazalo, da je od obravnavanih jezikov daleč najhitrejši razredni objektno-orientiran jezik java. Objektno-orientirani jeziki s prototipi so zaradi svoje dinamične narave, tako v delu z objekti kot tudi spremenljivkami ter dedovanjem, prisiljeni žrtvovati del svoje računske učinkovitosti. Omembe vredno pa je kljub temu dejstvo, da java pri izvajanju uporablja delno prevedeno vmesno kodo (angl. bytecode), medtem ko lua in javascript začneta z neprevedeno izvorno kodo (angl. source code).

Najbrž bi lahko rekli, da je java najbolj optimizirana od jezikov, saj na njej že dolgo delajo velika podjetja z visokimi standardi ter viri, s katerimi lahko financirajo razvoj jezika. Tudi javascript je močno optimiziran, saj je hitrost delovanja spleta, kjer je največ uporabljan, zelo pomembna za njegove uporabnike. Vseeno pa je javascriptov glavni namen prav raba na spletu, zato mu ekstremna optimizacija ponavljajočih se operacij, ki smo jih testirali mi, verjetno ni ravno prioriteta. Po drugi strani se java pogosto uporablja pri delu z večjimi količinami podatkov, zato je efektivnost njenega delovanja pri ponavljajočih se operacijah precej pomembna.

Standardna implementacija lue se v večini primerov odreže najslabše. To je po eni strani pričakovano – lua ni namenjena hitremu delu z veliko količino podatkov, s takimi nalogami se ponavadi obrne na C, s katerim lahko odlično sodeluje. Luin namen je predvsem, da služi kot dinamičen, majhen ter preprost skriptni jezik, ki zmore veliko že sam, lahko pa se enostavno poveže tudi z drugimi jeziki. To je tudi eden glavnih razlogov za nastanek projekta luaJIT – pohitritev ter optimizacija lue, da lahko tudi sama zaživi kot hiter in zmogljiv jezik, primeren za uporabo v večjih projektih. V vseh testih se luaJIT odreže občutno bolje od standardne lue in tako očitno doseže svoj cilj. Na sploh izgleda, da so JIT prevajalniki zmogljiva rešitev, ki je sicer dosti kompleksnejša od enostavnejšega sprotnega tolmačenja, a tudi dosti bolj učinkovita.

V testih se je v glavnem pokazalo, da učinkovita implementacija jezika ponavadi zmaga nad idejami in obliko jezika samega. Zanimiva izjema je test dinamičnega dodeljevanja različnih vrednosti spremenljivki, v katerem standardna implementacija lue prehiti javascript, ki je bil sicer občutno nad njo. Izgleda, da preprostejša implementacija objektov v tem primeru lui pomaga do zmage.

Poglavje 5

Zaključek

Dandanes so objektno-orientirani jeziki s prototipi precej redkejši od objektno-orientiranih jezikov, ki temeljijo na uporabi razredov. Javascript in lua sta praktično edina od bolj znanih jezikov, ki uporabljata prototipe in še za njiju obstaja veliko število raznih primerov in navodil, kako njuno delovanje približati razrednemu ali pa vsaj pokazati, da se od slednjega ne razlikuje veliko. Mnogo ljudi prototipne jezike uporablja, ne da bi se zavedali točne razlike med njimi in objektnimi. Na sploh izgleda, da sta se javascript ter lua prijela bolj zato, ker sta našla svoji niši uporabe, kot zato, ker uporabljata prototipni pristop k delu z objekti. Javascript je postal jezik spleta, lua pa vezni jezik za ostale jezike ter dodatek, ki lajša delo z njimi.

Prototipni pristop ima svoje prednosti. Objekte predstavi na oprimjemljivejši način, saj nam omogoča, da vedno delamo z dejanskimi objekti in ne potrebujemo razredov, ki bi delovali kot navodila zanje. Programe je v prototipnih jezikih mogoče pisati z manj predhodnega načrtovanja, saj lahko, kot smo pokazali v primerjavi dedovanja, objekte kadarkoli predelamo z dodajanjem ter odstranjevanjem komponent, brez, da bi bili prisiljeni za to napisati nov razred. Tudi o tipih spremenljivk nam ni potrebno posebej premišljevati, saj so ti v veliki večini objektno-orientiranih jezikov s prototipi dinamični in ista spremenljivka lahko po potrebi tudi zamenja svoj tip – kot

smo pokazali v testih hitrosti dinamičnega dodeljevanja tipov.

Prototipne jezike je smiselno uporabiti za manjše ali pa pogosto spreminjajoče se projekte. Dobro delujejo tudi kot skriptni jeziki in na sploh v vseh situacijah, ko kodo pišemo sproti brez veliko predhodnega premisleka in načrtovanja. Če naredimo napako ali pa se pojavi nov problem, ki ga prej nismo opazili, lahko z uporabo prototipov ter lažje prilagodljivih objektov svoj program hitreje popravimo ter nadaljujemo z delom. Tudi če se pojavijo robni primeri, kjer mora biti le en objekt rahlo drugačen od večine ostalih, ga je dosti lažje neposredno prilagoditi, kot pa zanj pisati lasten razred.

Prototipni jeziki se izkažejo za nekoliko slabše v primerih, v katerih je potrebne več računske učinkovitosti. Java se je v vseh naših testih hitrosti, v katere je bila vključena, izkazala kot najhitrejši od jezikov. Statično tipiziranje, klasično nasledstvo ter trše določeni objekti dajo razrednim jezikom prednost v hitrosti izvajanja pred prototipnimi ter jih s tem delajo primernejše za večje obdelave podatkov in zahtevnejše računske operacije, kakršne pogosto najdemo v večjih projektih. Zahtevajo nekoliko več načrtovanja, kar pa je pri večjih projektih že tako ali tako potrebno in ne predstavlja dodatnega problema. Izgleda tudi, da se objektno-orientirani jeziki z razredi bolje držijo principa enkapsulacije – java ima od obravnavanih jezikov daleč najbolje implementiran nadzor dostopov, kar je še dodatna prednost pri večjih projektih, kjer ima ta precej večji pomen.

Nekateri se prednosti prototipnega pristopa zavedajo – javascript je pogosto prikazan kot začetnikom prijazen jezik, saj je delo v njem brez potrebe po razmišljanju o tipih spremenljivk ter načrtovanju razredov dejansko lažje, kot na primer delo v javi. Vseeno pa izgleda, da so popularni objektno-orientirani jeziki s prototipi danes v rabi bolj zaradi ostalih dobrih lastnosti, ki jih imajo, kot pa zaradi svoje rabe prototipov ter poenostavitve dela, ki ju ti prinesejo s seboj.

Dodatek A

Koda programov za teste hitrosti izvajanja

A.1 Koda programov za testiranje učinkovitosti funkcijskih klicev

A.1.1 Java

Program A.1: Test učinkovitosti funkcijskih klicev v javi

```
public static int fibonacci(int stevilo) {
    if (stevilo == 0) {
        return 0;
    } else if (stevilo == 1) {
        return 1;
    } else {
        return     fibonacci(stevilo - 1)
                 + fibonacci(stevilo - 2);
    }
}
```

A.1.2 Javascript

Program A.2: Test učinkovitosti funkcijskih klicev v javascriptu

```
function fibonacci(stevilo) {
    if (stevilo == 0) {
        return 0;
    } else if (stevilo == 1) {
        return 1;
    } else {
        return    fibonacci(stevilo - 1)
                + fibonacci(stevilo - 2);
    }
}
```

A.1.3 Lua

Program A.3: Test učinkovitosti funkcijskih klicev v lui

```
fibonacci = function(stevilo)
    if stevilo == 0 then
        return 0
    elseif stevilo == 1 then
        return 1
    else
        return (fibonacci(stevilo - 1) +
                fibonacci(stevilo - 2))
    end
end
```

A.2 Koda programov za testiranje dedovanja

A.2.1 Java

Program A.4: Dedovanje v javi

```
class ClassA {
    public int funA(int number) {
        return number + 1;
}   }

class ClassB extends ClassA {
```

```
    public int funB(int number) {
        return number + 2;
}   }

class ClassC extends ClassB {
    public int funC(int number) {
        return number - 1;
}   }

class ClassD extends ClassC {
    public int funD(int number) {
        return number - 2;
}   }

class TestClass extends ClassD {
    int[] numArr;
    int testVar;

    public TestClass(int arg) {
        this.numArr = new int[arg];
        for (int i = 0; i < arg; i++) {
            this.numArr[i] =
                (int)Math.floor(Math.random() * 4);
    }   }

    public void runTest(int arg) {
        for (int i = 0; i < arg; i++) {
            switch(this.numArr[i]) {
                case 0: this.testVar =
                            this.funA(this.testVar);
                            break;
                case 1: this.testVar =
                            this.funB(this.testVar);
                            break;
                case 2: this.testVar =
                            this.funC(this.testVar);
                            break;
                default: this.testVar =
                            this.funD(this.testVar);
                            break;
}   }   }   }
```

A.2.2 Javascript

Program A.5: dedovanje v javascriptu

```
var objA = Object.create(Object);
objA.funA = function(number) {return number + 1};

var objB = Object.create(objA);
objB.funB = function(number) {return number + 2};

var objC = Object.create(objB);
objC.funC = function(number) {return number - 1};

var objD = Object.create(objC);
objD.funD = function(number) {return number - 2};

var testObj = Object.create(objD);
testObj.numArr = numArr;
testObj.testVar = 0;
testObj.runTest = function() {
    for (var i = 0; i < this.numArr.length; i++) {
        switch (this.numArr[i]) {
            case 0: this.testVar =
                        this.funA(this.testVar);
                        break;
            case 1: this.testVar =
                        this.funB(this.testVar);
                        break;
            case 2: this.testVar =
                        this.funC(this.testVar);
                        break;
            default: this.testVar =
                        this.funD(this.testVar);
                        break;
}   }   }
```

A.2.3 Lua

Program A.6: Dedovanje v lui

```
objA, objB, objC, objD, testObj = {}, {}, {}, {}, {}
objA.funA = function(this, stevilo)
    return stevilo + 1
end

setmetatable(objB, {__index = objA})
objB.funB = function(this, stevilo)
    return stevilo + 2
end
```

```
setmetatable(objC, {__index = objB})
objC.funC = function(this, stevilo)
    return stevilo - 1
end

setmetatable(objD, {__index = objC})
objD.funD = function(this, stevilo)
    return stevilo - 2
end

setmetatable(testObj, {__index = objD})
testObj.numArray = numArr
testObj.testVar = 0
testObj.runTest = function(this)
    for i = 1,arg[1] do
        if this.numArray[i] == 1 then
            this.testVar = this:funA(this.testVar)
        elseif this.numArray[i] == 2 then
            this.testVar = this:funB(this.testVar)
        elseif this.numArray[i] == 3 then
            this.testVar = this:funC(this.testVar)
        else
            this.testVar = this:funD(this.testVar)
        end
    end
end
```

A.3 Koda programov za testiranje pošiljanja sporočil

A.3.1 Java

Program A.7: Pošiljanje sporočil v javi

```
// klici v metodi main
ClassA1 objA = new ClassA1();
ClassB1 objB = new ClassB1();
objA.setPair(objB);
objB.setPair(objA);
objA.runTest(stevilo);

//definicije razredov
class ClassA1 {
    int valA;
```

```
    ClassB1 pair;

    public ClassA1() {
        this.valA = 0;
    }

    public void setPair(ClassB1 pair) {
        this.pair = pair;
    }

    public void send() {
        this.pair.receive(this.valA+1);
    }

    public void receive(int valArg) {
        this.valA = valArg;
    }

    public void runTest(int argNum) {
        while(this.valA < argNum) {
            this.send();
        }
        System.out.println(this.valA);
    }
}

class ClassB1{
    int valB;
    ClassA1 pair;

    public ClassB1() {
        this.valB = 0;
    }

    public void setPair(ClassA1 pair) {
        this.pair = pair;
    }

    public void send() {
        this.pair.receive(this.valB+1);
    }

    public void receive(int valArg) {
        this.valB = valArg;
        this.send();
    }
}
```

A.3.2 Javascript

Program A.8: Pošiljanje sporočil v javascriptu

```
var objA = Object.create(Object);
var objB = Object.create(Object);
objA.valA = 0;
objB.valB = 0;
objA.pair = objB;
objB.pair = objA;

objA.send = function() {
    this.pair.receive(this.valA+1);
}

objA.receive = function(valArg) {
    this.valA = valArg;
}

objB.send = function() {
    this.pair.receive(this.valB+1);
}

objB.receive = function(valArg) {
    this.valB = valArg;
    this.send();
}

objA.runTest = function() {
    while(this.valA < stevilo) {
        this.send();
    }
    console.log(this.valA);
}

objA.runTest();
```

A.3.3 Lua

Program A.9: Pošiljanje sporočil v lui

```
objA, objB = {}, {}
objA.valA = 0
objB.valB = 0
```

```
objA.pair = objB
objB.pair = objA

objA.send = function(this)
    this.pair:receive(this.valA+1)
end

objA.receive = function(this, valArg)
    this.valA = valArg
end

objB.send = function(this)
    this.pair:receive(this.valB+1)
end

objB.receive = function(this, valArg)
    this.valB = valArg
    this:send()
end

objA.runTest = function(this)
    while(this.valA < tonumber(arg[1])) do
        this:send()
    end
    print(this.valA)
end

objA:runTest()
```

A.4 Koda programov za testiranje dinamičnega dodeljevanja vrednosti

A.4.1 Javascript

Program A.10: Dinamično dodeljevanje vrednosti v javascriptu

```
var testObj = Object.create(Object);
testObj.testVar = null;
var dummyObject = Object.create(Object);

testObj.runTest = function(args) {
    for (var i = 0; i < args; i++) {
        switch (i % 5) {
            case 0: this.testVar = 123;
                    break;
```

```
            case 1: this.testVar = "Hello, world!"
                    break;
            case 2: this.testVar = 0.123456;
                    break;
            case 3: this.testVar = dummyObject;
                    break;
            default: this.testVar = null;
                     break;
        }
    }
}
```

A.4.2 Lua

Program A.11: Dinamično dodeljevanje vrednosti v lui

```
testObj = {}
testObj.testVar = nil
dummyObject = {}

testObj.runTest = function(this)
    for i = 1,arg[1] do
        if i % 5 == 0 then
            this.testVar = 123;
        elseif i % 5 == 1 then
            this.testVar = "Hello, world!"
        elseif i % 5 == 2 then
            this.testVar = 0.123456;
        elseif i % 5 == 3 then
            this.testVar = dummyObject;
        else
            this.testVar = nil;
        end
    end
end
```

Literatura

[1] Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölzle, John Maloney, Tobias Pape, Randall B. Smith, David Ungar, and Mario Wolczko. Self handbook for self 4.5.0. `http://handbook.selflanguage.org/4.5/`. Dostopano: 2016-07-31.

[2] Deborah J. Armstrong. The quarks of object-oriented development. *Commun. ACM*, 49(2):123–128, February 2006.

[3] Dave Atchley. Understanding prototypes, delegation and composition. `http://www.datchley.name/understanding-prototypes-delegation-composition/`. Dostopano: 2016-08-15.

[4] Bob Bemer. Simula, an algol offspring – first object-oriented programming language. `http://www.bobbemer.com/SIMULA.HTM`. Dostopano: 2016-03-29.

[5] Günther Blaschek. *Object-oriented Programming: With Prototypes*. Springer Science & Business Media, 2012.

[6] W3 comunity. A short history of javascript. `https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript`. Dostopano: 2016-03-29.

[7] Christophe Dony, Jacques Malenfant, and Daniel Bardou. Classifying prototype-based programming languages. *Prototype-based Programming: Concepts, Languages and Applications*, 86, 1998.

[8] Christophe Dony, Jacques Malenfant, and Pierre Cointe. Prototype-based languages: From a new taxonomy to constructive proposals and their validation. *SIGPLAN Not.*, 27(10):201–217, October 1992.

[9] Wikibooks editors. Javascript/access control. `https://en.wikibooks.org/w/index.php?title=JavaScript/Access_control&oldid=3076604`. Dostopano: 2016-07-31.

[10] David Flanagan. *JavaScript the definitive guide*. O'Reilly Media, Inc., 2006.

[11] Google. Introduction to chrome v8. `https://developers.google.com/v8/intro/`. Dostopano: 2016-08-20.

[12] Jian Huang. A brief history of object-oriented programming. `http://web.eecs.utk.edu/~huangj/CS302S04/notes/oo-intro.html`. Dostopano: 2016-03-29.

[13] Roberto Ierusalimschy. *Programming in Lua, First Edition*. Lua.Org, 2003.

[14] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of lua. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 2–1–2–26, New York, NY, USA, 2007. ACM.

[15] IETF. The javascript object notation (json) data interchange format. `https://tools.ietf.org/html/rfc7159`. Dostopano: 2016-08-15.

[16] Alan Kay. Dr. Alan Kay on the meaning of "object-oriented programming". `http://www.purl.org/stefan_ram/pub/doc_kay_oop_en`. Dostopano: 2016-03-29.

[17] Online Historical Encyclopaedia of Programming Languages. Kevo, stack and prototype-based object-oriented language. `http://`

hopl.info/showlanguage.prx?exp=1691&language=Kevo. Dostopano: 2016-07-31.

[18] Oracle. Java documentation (se 8). https://docs.oracle.com/javase/8/. Dostopano: 2016-07-31.

[19] Oracle. Oracle jdk 7 and jre 7 certified system configurations. http://www.oracle.com/technetwork/java/javase/config-417990.html. Dostopano: 2016-08-28.

[20] Mike Pall. Luajit frequently asked questions. http://luajit.org/install.html. Dostopano: 2016-08-31.

[21] The GNOME Project. Gnome system monitor. https://github.com/GNOME/gnome-system-monitor. Dostopano: 2016-08-31.

[22] Tim Rentsch. Object oriented programming. *SIGPLAN Not.*, 17(9):51–57, September 1982.

[23] Antero Taivalsaari. Classes vs. prototypes - some philosophical and historical observations. In *Journal of Object-Oriented Programming*, pages 44–50. SpringerVerlag, 1996.